The Sonar Simulation Toolset, Release 4.1: Science, Mathematics, and Algorithms

by Robert P. Goddard

Technical Report
APL-UW TR 0404
March 2005

Applied Physics Laboratory University of Washington
1013 NE 40th Street Seattle, Washington 98105-6698

ONR Contracts N00014-01-G-0460 and N00014-98-G-0001

Acknowledgments

The Sonar Simulation Toolset was developed with sponsorship from several U. S. Navy sources, most recently ONR Code 333 (Adam Nucci) and ONR's "ARL Project" (Code 321US, John Tague). Earlier sponsors included the Naval Surface Warfare Center, Carderock (William Beatty) and the Naval Underwater Systems Center, New London, CT (Walter Hauck and Thomas Wheeler).

The SST development team at APL-UW consists of the author, Pete Brodsky, Patrick Tewson, Brandon Smith, and Don Percival. Warren Fox, Jim Luby, and Chris Eggen have provided guidance, testing, and leadership. Earlier team members included Beth Kirby, Kou-Ying Moravan, Megan Hazen, Bill Kooiman, Gordon Bisset, and undergraduates Pat Lasswell and Jason Smith. Kou-Ying Moravan and Pierre Mourad supplied the original Fortran implementations from which several of the boundary models started. Mike Boyd, Warren Fox, Greg Anderson, and Chris Eggen have contributed CASS expertise. The REVGEN project, where SST got its start, was led by Dave Princehouse.

The Comprehensive Acoustic System Simulation (CASS) program is available through the support of the Naval Undersea Warfare Center, Newport (NUWCDIVNPT). Permission to use and distribute CASS and GSM is granted by NUWC (Emily McCarthy).

SST's users are the ingredients that make SST a useful tool instead of an academic exercise. Special thanks are due to the many SST users at ARL/PSU and NUWC (Newport, RI), with whom we have enjoyed a long and fruitful collaborative relationship.

Typeset August 20, 2004

Abstract

The Sonar Simulation Toolset (SST) is a computer program that produces simulated sonar signals, enabling users to build an artificial ocean that sounds like a real ocean. Such signals are useful for designing new sonar systems, testing existing sonars, predicting performance, developing tactics, training operators and officers, planning experiments, and interpreting measurements. SST's simulated signals include reverberation, target echoes, discrete sound sources, and background noise with specified spectra. Externally generated or measured signals can be added to the output signal or used as transmissions. Eigenrays from the Generic Sonar Model (GSM) or the Comprehensive Acoustic System Simulation (CASS) can be used, making all of GSM's propagation models and CASS's Gaussian Ray Bundle (GRAB) propagation model available to the SST user. A command language controls a large collection of component models describing the ocean, sonars, noise sources, targets, and signals. The software runs on several different UNIX computers. The software runs on several UNIX computers and Windows. SST's primary documentation is the SST Web (a large HTML "web site" distributed with the SST software), supported by a collection of documented examples.

This report emphasizes the science, mathematics, and algorithms underlying SST. This report is intended to be updated often and distributed with SST as an integral part of the SST documentation.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

The Sonar Simulation Toolset (SST) is a computer program that produces simulated sonar signals as "heard" by a user-specified active or passive sonar in a user-specified ocean environment. It enables a user to create an "artificial ocean" that may be used to test new or proposed sonar systems or tactics, to train sonar operators, to plan experiments, or to validate models of underwater acoustic phenomena by comparing simulation results with measurements.

This paper focuses on the science, mathematics, and algorithms used in SST. It is intended, in part, as a tutorial to introduce scientists and engineers to the technical issues that must be addressed by any sonar signal simulation system. It is also intended as a supplement to the existing SST documentation, adding information on "how it works" and "why" to the existing descriptions of "how to do it". And it is intended to help scientists, engineers, trainers, and technical managers decide whether SST might be useful in their projects.

This report started as an on-line, extended version of a paper for the classified US Navy Journal of Underwater Acoustics. To enable wider distribution, that paper's sensitive "Applications" section is omitted here. Because it is intended to be used as an integral part of the SST documentation, this report contains software-specific details that were omitted from the JUA version. It contains several features to enhance on-line browsing, including active (click-through) references to equations, figures, related text sections, and citations. This report will be updated often as the software changes.

A separate on-line document, the SST Web [SST Web], gives details about how to use SST. It includes working examples of SST simulations for several different kinds of active and passive sonar systems. Yet another on-line document, which is generated from the source code and comments therein using the Doxygen [Doxygen] software, covers the internal design of SST.

The SST software, together with all of the documentation just mentioned (including this report), are delivered to DoD agencies and contractors via a secure Web site [APL SBU].

We will assume that the reader is familiar with underwater sound at the level of [Urick 1983], and with digital sonar processing at the level of Knight *et al.* [Knight

1981]. It will be helpful, but not required, for the reader to be familiar with discrete-time signal processing [Oppenheim Schafer 1989] and the fundamentals of object-oriented software design [Page-Jones 2000].

1.2 Objectives and Attributes

SST is certainly not the only sonar simulation system available to the Navy community. It may or may not be appropriate for a given application, depending on the purpose of the simulation. General modeling tools like CASS [Weinberg et al. 2001] or SWAT [Sammelmann 2002], real-time simulators like WAF [Correia 1988, Katyl 2000], tactical simulators like TRM, or application-specific tools may have a place in a simulation tool kit. To understand where SST fits, consider the following objectives and attributes of SST:

Signal Level Simulation: SST produces sound, suitable as input for users' ears, for the front end of a sonar system, or for a computer model of an existing or proposed sonar front end. It does not produce plots or predict performance by itself.

Portable: SST is designed to run on nearly any modern general-purpose computer. Each distribution includes pre-built versions for SPARC/Solaris, Intel/Linux, and Intel/Windows/Cygwin [Cygwin] systems, plus source code and porting tools to make it as easy as possible to port it to other systems.

General: SST is suitable for simulating a wide variety of active or passive sonar systems, sound sources, and targets in many different environments and scenarios. Multiple sound sources, multiple targets, complicated sonar systems and signals, arbitrary trajectories, variable bathymetry, various surface and bottom models, and many other details can be specified using SST's flexible command language.

Broadband: SST is suitable for signals of any bandwidth. The frequency range is limited primarily by its ray-based propagation models, which can be useful as low as a few kHz in shallow water or even lower in deep water, depending on the requirement for fidelity.

Multistatic: There can be any number of sound sources, receivers, and targets on any number of platforms.

Multi-channel: The sonar can have any number of channels, each of which is characterized by its own sensitivity pattern and offset. A channel may represent the

signal behind one transducer (element-level) or for one beam behind the beam former (beam-level). Correlations between channels are carefully controlled.

Non-Real-Time: SST is not constrained to run in real time.

Flexible Fidelity: SST's lack of real-time constraint allows the user to trade off speed for fidelity and detail, striking whatever balance is consistent with the user's requirements, budget, and patience. The idea is to support whatever level of fidelity and detail is required for each application, without unnecessarily slowing the simpler simulations. The level of realism can, if necessary, be quite high, and simple simulations can achieve much faster than real-time throughput.

Embeddable: SST may be used as a signal generation component within a higher-level simulation. It is being used that way within TRM at ARL-PSU.

Streamable: SST's results are produced in time order. The early parts of the output are written as soon as possible, usually well before the simulation is finished. This feature is essential for embedded applications and for parallel processing (currently in development), and it helps minimize memory requirements for long runs. Another advantage is that users can examine partial results before deciding whether to continue a long simulation run.

Object Oriented: SST's command language, its primary implementation language (C++), and SST's design are organized around the concepts of *objects*, *classes*, *encapsulation*, *inheritance*, and *polymorphism*. The advantages of this approach are documented in standard software engineering texts [Page-Jones 2000].

Unclassified: SST's distribution is limited to DoD and DoD contractors only (critical technology), but the code and documentation are not classified.

1.3 History

Most of the support for SST's development has come through projects supporting specific applications. Hence SST's current capabilities reflect, to a dominant degree, the union of the requirements specified for those applications. In response to changing requirements, innovations were introduced in roughly the following order:

- SST's roots trace back to the REVGEN (Reverberation Generator) [Princehouse 1975,Princehouse 1978,Goddard 1986] project of the 1970s and early 1980s. SST itself started in 1989. The focus was on high-frequency, narrow-band, monostatic active sonar systems. The initial few versions generated reverberation and target echoes using a narrow-band point scatterer model (Sec. 10).

- In the early 1990s the emphasis shifted to broadband sonars. This led to updated high-frequency environmental models [APL Models 1994], an optional scattering-function reverberation model [Luby Lytle 1987], frequency-dependent environmental models and beam patterns, and the "data flow" architecture (Sec. 4.3).

- Another thrust during the middle 1990s was toward multistatic operation. SST's monostatic reverberation models were replaced by a fully bistatic, broadband one based on scattering functions (Sec. 10). Bistatic surface and bottom models were also introduced (Sec. 6).

- During this period, both the number of SST users and the size and complexity of the code rose dramatically. In response, we replaced the earlier text documentation with an extensively cross-linked HTML Web version [SST Web], and C++ replaced C as the primary programming language.

- Starting in 1997 several applications involved frequencies below 10 kHz. This led to a new mid-frequency bistatic surface model [Gilbert 1993], an upgraded bottom model, an external target model (Sec. 9.1.3), and better support for passive sonars.

- The Navy's concern shifted from deep water to littoral environments. In response, we added support in SST for range-dependent propagation using eigenrays computed using the GRAB [Weinberg Keenan 1996] eigenray model.

- In broadband, shallow-water applications, SST users recognized that the processing gain for simulated target echoes versus reverberation and countermeasures was too optimistic. The main culprit was identified as near-specular scattering from the surface and bottom. In response, we added time and frequency spreading (reduced coherence in frequency and time) (Sec. 11).

- Several users wanted to use SST as a signal generation component in a higher-level tactical simulation. Other users wanted to use some of SST's models in other environments. These requirements have led to a continuing push toward interoperability.

- Current efforts, driven by user concerns, include improved coherence control, faster element-level simulations for systems with a very large number of hydrophones, user interface support for combined active and passive processing, ship wakes, very long active transmissions, and better realism at lower frequencies.

- Everyone always wants more speed, better documentation, fewer software errors, a simpler and more intuitive user interface, more checks to prevent user

mistakes, interoperability with other systems, and more and better support for users. We are constantly working to improve SST and our services in all of these measures.

SST has played a significant role in many studies over the years. Unclassified published ones include [Eggen Goddard 2002], [Goddard 2000], and [Rouseff et al. 2001].

1.4 Release

The version of SST described here is Release 4.1, dated November 2002.

1.5 Outline

Section 2 is a general overview of SST and its component models. Section 3 presents a simple example showing the results of an SST simulation. The bulk of the paper describes the science and mathematics underlying SST. The order in which SST's various sub-models are described is intended to support linear reading by placing the background needed to understand a model's requirements before that model's description. We conclude, in Section 11, with a brief description of current and planned projects to improve SST's realism, scope, and ease of use.

1.6 Notation

In nearly all of this paper, we treat the signals, transformations, and models as *continuous* in time, space, frequency, and direction. This is a physicist's point of view, not a software engineer's. Of course, as in any digital implementation, conceptually continuous functions are sampled at discrete values of their independent variables, and integrations are implemented as finite sums, and precision is limited. We discuss digital samples when it is necessary to do so. In our view, however, the physical and mathematical concepts are easier to understand in the continuous domain, so we remain there whenever possible. The mapping between continuous and digital domains is covered well by standard texts [Oppenheim Schafer 1989, Hamming 1973].

SST is object oriented on two levels, the SST command language and the implementation language. For the most part, an SST input file consists of statements that define objects that are instances of built-in SST classes, and assign values to named

attributes (parameters) of those objects. Each of the classes visible through the command language is implemented in terms of a C++ class having the same name, with attributes corresponding to C++ member variables. We do not distinguish here between the two kinds of classes.

Within paragraph text, class names are displayed in bold *sans serif* font and capitalized; examples are **Signal**, **PistonBeam**, and **JacksonBottom**. Names of class attributes are displayed in slanted *sans serif* font and uncapitalized; examples are *frequency*, *radius*, and *soundSpeedRatio*.

2 Overview

2.1 Assumptions

The simulated sound produced by SST consists of a digital representation of the predicted signal in each channel of the sonar receiver's processing path. This sound may contain components from four types of sources:

Discrete sound sources: "Passive targets," such as ships or countermeasures, that radiate noise from a compact region

Diffuse sound sources: Environmental noise and self noise

Discrete scatterers: "Active targets," such as submarines or rocks, that echo an active sonar's transmitted pulses (or other sound) back to the sonar receiver

Reverberation: Diffuse scatterers, such as the ocean bottom, that send back many overlapping echoes of an active sonar's pulse

The sonar system can be passive (receiver only), or it can be active (listening for echoes of its own transmissions). Active sonars can be monostatic (transmitter and receiver on the same platform), bistatic (transmitter and receiver on two different platforms), or multistatic (employing several transmitters or receivers).

The receiver can have any number of channels (transducers or beams). There can be any number of sound sources and targets, and any number of sound paths (eigenrays) connecting sources, scatterers, and receivers. For both passive and active sonars, the signals can have an arbitrarily wide range of frequencies. The ocean, sound scatterers, and beam patterns act as filters that alter the frequency content of

the sound. Transmissions and listening intervals can be arbitrarily long. Of course, all of these parameters will have an impact on size and speed, so the practical limits depend on available disk space, memory, processor speed, and the patience of the user.

Over the last decade, SST's most active users (and most of its financial support) have come from the torpedo development community. Therefore, its high-frequency environmental models are more complete and up-to-date than its lower-frequency models. The current propagation models are based on eigenrays, not on direct solutions of the wave equation. These factors limit the realism of SST's simulations for low frequencies, especially in shallow water.

2.2 Outputs

SST's primary output is multi-channel sampled sound: a digital representation of the simulated signal somewhere in the sonar receiver. A channel can represent either the signal behind a transducer (for element-level simulations) or an output of the beamformer (for beam-level simulations).

SST can also produce several other types of data, including the reverberation scattering function, various types of spectra and cross-spectra, and synthesized signals intended for use as transmissions or noise sources. Some of these are intermediate products used in the simulation, and some result from general-purpose signal processing tools (like a spectrum analyzer) that the user can apply to any signal.

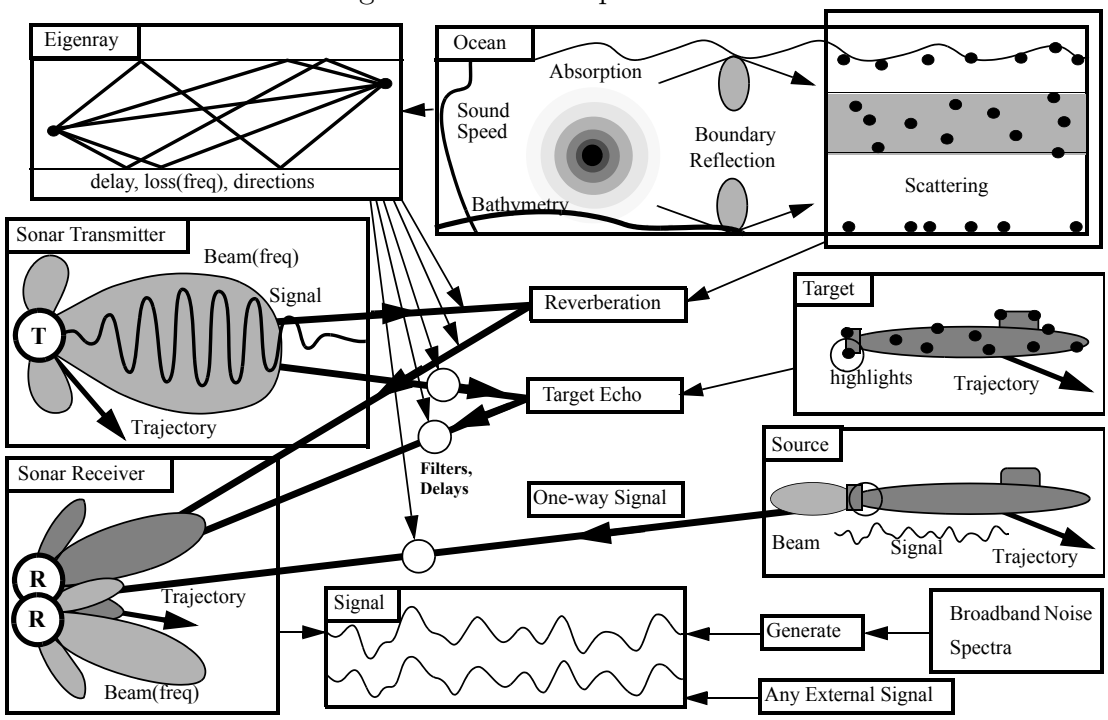
These simulated signals and related data can be written into an external file in any of several user-specified forms, binary or text. Any of these forms can be read by SST as well as written. The ways that SST can represent or store signals are described in Sec. 4.

SST does not produce plots or other displays. Post-run analysis and plotting can be done using commercial packages like Mathematica [Mathematica] or Matlab [Matlab], or public-domain tools like Octave [Octave] or Gnuplot [Gnuplot], specialized tools like the SIO package from Scripps [Hodgkiss 1989], or many other tools. A large and growing set of useful Matlab scripts are provided with SST.

2.3 Inputs

The primary inputs to an SST simulation consist of commands for generating the various types of signals, plus assignment statements in which the user specifies the

Figure 1: SST Component Models



characteristics of the ocean environment, the sonar transmitter and receiver, active and passive targets, and the format of the simulated signal. These specifications and commands are expressed in a simple but flexible language with an object-oriented flavor reminiscent of Python or C++. They may be entered from the keyboard, read from text files, or passed via a pipe from a higher-level program. The language supports user-defined variables and comments to help make the scripts readable.

Signals used as transmissions or noise sources can come from external files having the same file formats as SST's outputs. SST also includes tools for specifying and generating these signals internally (Sec. 4.6).

Eigenrays and beam patterns can come from external files, text or binary.

2.4 Models

SST is based on a large number of underlying *models* — mathematical representations of physical phenomena — which specify how each element of the environment affects the sound. Figure 1 summarizes those component models and the relationships among them.

Eigenray Model: SST's sound propagation models are based on *eigenrays* — paths through the water along which sound propagates between two specified endpoints. Sound in the neighborhood of a receiver or scatterer is a sum of copies of the original transmitted sound, each of which arrives at a different time from a different direction, and each of which has been attenuated in a frequency-dependent way. SST provides three choices (plus variants) for eigenray models: a straight-line model with reflections (the base class), one based on eigenrays from the GRAB eigenray model in the CASS software, and one based on any of several eigenray models provided by the GSM software. The eigenray models are discussed in Sec. 5.

Ocean Model: The SST user specifies the ocean environment by describing characteristics of the surface and bottom, the depth, the volume scattering strength, and the sound speed and absorption rate of the water itself. These models act as inputs to the eigenray model and the reverberation model. The various components of the ocean model are discussed in Sec. 6.

Sonar and Source Models: The sonar receiver is specified by giving its trajectory through the water, the location of each channel's phase center, and the beam pattern giving each channel's sensitivity versus direction and frequency. Each discrete sound source is described in terms of its trajectory, the signal it transmits, and the directional behavior of the transmission (beam patterns). An active sonar's transmitter is treated just like any other sound source; hence the "Sonar Transmitter" model in Fig. 1 is the same as the "Source" model. These models are discussed in Sec. 7.

Target Model: Each target (for active sonars) is an object (with a trajectory) that receives a signal from a source and re-transmits it to a sonar; hence it acts like a group of receivers and sources back to back. Several target models are available; each one provides a different way to specify the relationship between the received sound and the re-transmitted sound. The current target models describe that relationship in terms of *highlights*. They are described in Sec. 9.1.

Direct (One-way) Sound Propagation: The heavy arrows in Fig. 1 represent the various ways that a source signal is transformed on its way to being received by a sonar system. The simplest one, marked "One-way Signal" on the diagram, is also called "direct" sound propagation. This transformation combines properties of the source model, the eigenray model, and the sonar model, and reduces them to a set of filters and delays that transform the signal emitted by a source into the signal received by a sonar via paths that may involve reflection and refraction but not scattering. This transformation is described in Sec. 8.

Target Echoes: The heavy arrows in Fig. 1 marked "Target Echo" represent sound that scatters from a discrete target. It is essentially two "one-way signals" in series, with the target model in between. This transformation is described in Sec. 9.

Reverberation: The heavy arrows in Fig. 1 marked "Reverberation" represent sound that scatters from a very large number of very small scatterers distributed throughout the environment. This is conceptually very similar to a lot of target echoes, but SST treats them statistically, so the implementation of the transformation is quite different from target echoes. Reverberation is described in Sec. 10.

Noise and Other Sound: Sound from any source can be added to SST's one-way signals, target echoes, and reverberation. SST can produce broadband Gaussian noise having a user-specified power spectrum, as described in Sec. 4.6.1; this type of noise is useful to represent ambient noise, self-noise, and electronic noise. Broadband Gaussian noise can also be used as the emission from any sound source, for which it may be combined with harmonic families of tones or active transmit pulses. Signals for any of these purposes may be generated internally by SST, or read in from an externally generated file.

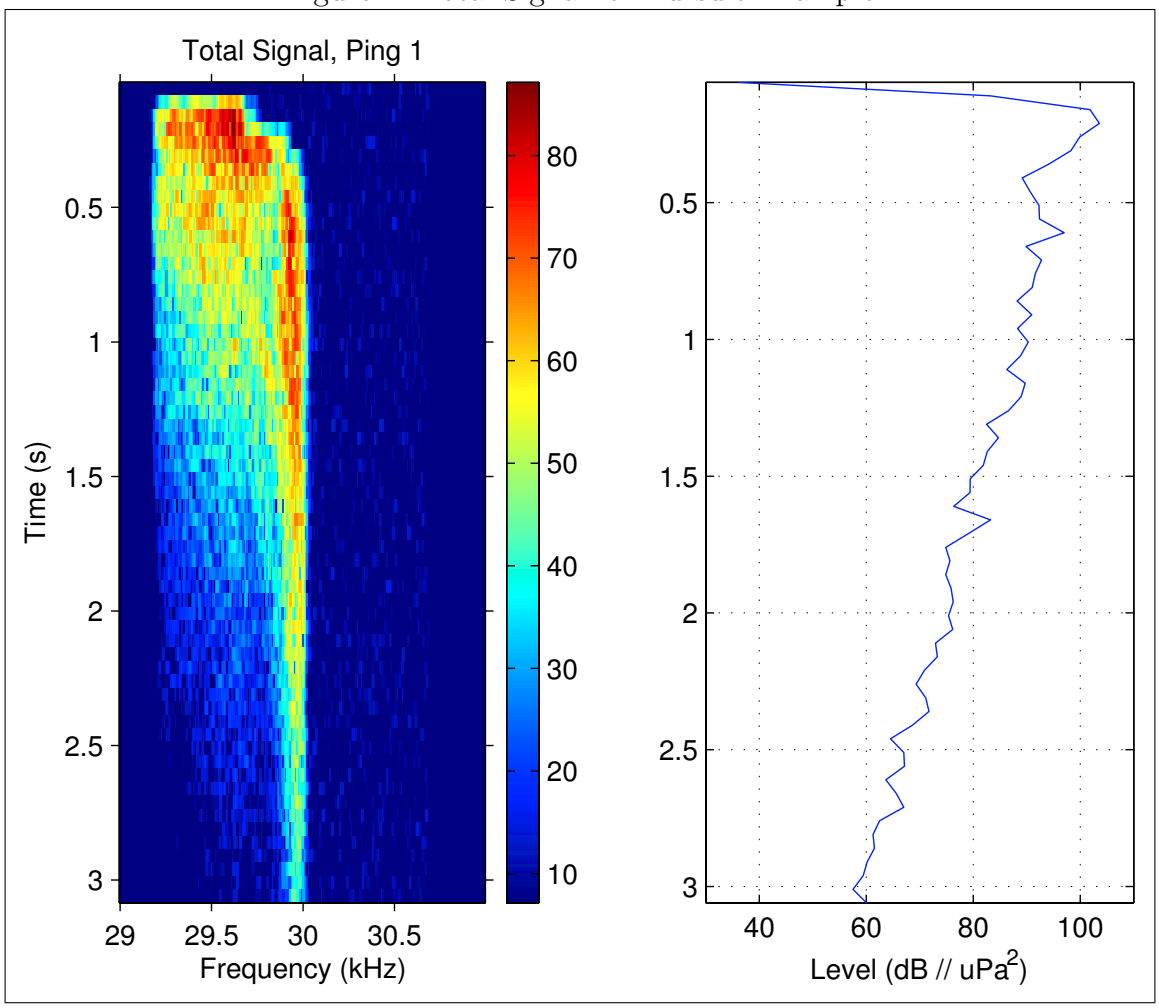
Signal Processing: All of the signal transformations are built on top of a diverse set of general-purpose signal processing tools for generating, summing, filtering, delaying, scaling, Fourier analyzing, and otherwise transforming signals. All of these signal processing tools can be hooked together like plumbing, or they can be used separately. The "data flow" architecture that makes this flexibility possible is described in Sec. 4.3, and the components themselves are described in Sec. 4.4.

2.5 Units and Coordinate Systems

In SST and in this paper, acoustic pressure is expressed in micro-Pascals (μPa), acoustic intensity is in μPa^2 or in dB//μPa^2, and angles are in degrees. All other measurements are expressed in MKS units.

In the *Earth-centered* coordinate system, vector components are in the order (North, East, Down) with an origin at an arbitrary point on the surface of the ocean. Earth curvature is ignored, except insofar as it is included in the CASS and GSM eigenray models (Sec. 5). In *platform-centered* coordinate systems (used for beam patterns, element offsets, and target highlights), vector components are in the order (Forward, Starboard, Below) relative to an arbitrary origin on the platform (sonar, source, or target). The mapping between the two types of coordinate systems is defined by the *trajectory* attribute of the vehicle, as described in Sec. 7.1.

Figure 2: Total Signal for Pursuit Example



2.6 Computers

This release of SST has been tested on SPARC/Solaris and Intel/Linux computers. An Intel/Windows port under the Cygwin [Cygwin] environment is planned. SST is designed to be portable, and previous versions of SST have been ported to several other UNIX-like systems.

3 Example

Figure 2 is a Matlab display showing the result of SST's "Pursuit" example, which is one of the standard examples provided with SST. The Pursuit scenario features a monostatic, high-speed, torpedo-like active sonar chasing after a submarine-like target. SST produced the multi-channel signal heard by the sonar, which includes the target echo, reverberation, and noise produced by the fleeing target. The Matlab scripts that produced the figure sliced that signal into short segments, computed the power spectral density for each segment, and displayed the results, both as a time-dependent spectrum and as total signal level.

The dominant feature in the figure is the reverberation, which comes primarily from the surface in this scenario. The early part of the signal comes from almost directly overhead, via the sidelobes of the beam pattern; hence it has very little Doppler shift. The later part of the reverberation comes from the main lobe, almost straight ahead; hence it has a high upward Doppler shift (the main ridge down the middle of the spectrum). This obvious "reverberation hook" from low early Doppler to high late Doppler is characteristic of forward-looking sonars. Less obvious is the streak that starts at about 0.9 seconds and merges with the main ridge; that comes from a path that reflects from the bottom.

The target echo is the blip just after 1.0 second and just above 29.5 kHz. It has a lower Doppler shift than the main reverberation ridge because it is running away. This is an "easy" high-Doppler detection despite the fact that the level of the target return is well below the level of the reverberation (the target doesn't show at all on the level plot on the right).

Note that this is a very simple analysis of a very simple scenario. Only one channel is shown (although the receiver in the example has 5 channels); complications like inter-channel correlations are in the SST-generated signal but not shown by this analysis. The pulse is a narrow-band pure tone, so the Fourier analysis used here is nearly optimal; a broadband pulse would require more complex analysis, probably featuring replica correlation. The sonar is monostatic, and only a few eigenrays are significant. These limitations are not imposed by SST, which can support much more complicated scenarios requiring much more complicated processing, both by SST and by the post-processing algorithms needed to analyze the results.

The SST script for this example can run in a few seconds on a modern workstation — close to real-time throughput (but not real-time in terms of response deadlines). More complicated scenarios would take more time.

4 Signals and Signal Transformations

The inputs, outputs, and much of the intermediate data of SST consist of *signals*. In this section we address the question, "What is a signal?" from several points of view, including abstract meaning, sampled representations, and software components. These issues are addressed early because they form the scaffolding on which much of the SST system is built.

4.1 Signal Representations

Conceptually, a signal $x_c(t)$ is a continuous, band-limited, multi-channel function of time t, with channels indexed by c. A signal may represent (for example) voltage on a set of wires or sound pressure at a set of locations in the water. Within SST each signal is represented as a sequence of sets of sample values $x_c(t_n)$ corresponding to discrete values t_n of the time. SST supports three distinct, equivalent representations of a signal: real samples, complex envelope, and windowed frequency domain. All signals in SST, whether they are used for input, output, or in between, are represented in one of these three ways.

4.1.1 Real Samples

The most straightforward way to represent a real signal $x_c(t)$ in SST is as a sequence of real samples on a uniform grid of time values:

$$x_c(t_n) = x_c(t_0 + nh), \tag{1}$$

where n is an integer and h is the time increment between samples.

If the Fourier transform of the original continuous signal $x_c(t)$ is negligible for frequencies greater than some maximum frequency $F_{\max}$, and the sample interval h is chosen such that $F_{\max}$ is within the Nyquist band:

$$h < 1/(2F_{\max}), \tag{2}$$

then the original signal can be recovered (in principle) using band-limited interpolation:

$$x_c(t) = \sum_{n=-\infty}^{\infty} \left(\frac{\sin(\pi(t - t_n)/h)}{\pi(t - t_n)/h} \right) x_c(t_n). \tag{3}$$

The problematic infinities and long decay time of this equation will be dealt with in Section 4.5.1, when we describe the delay algorithm.

SST Class: The base class from which all time-domain signals are derived is **Signal**, which has attributes *frequency*, *isComplex*, and *times*. To specify the real sample representation in any class derived from **Signal**, set *isComplex* to *false* and *times* to a **UniformGrid** specifying the sample times. More details about this family of classes will be given in the subsections 4.3 and 4.4 below.

4.1.2 Complex Envelope

For a *band-limited* signal, a bandwidth W and a center frequency F exist such that the Fourier transform of the signal is negligibly small outside the frequency range $F_{\min} = F - W/2$ to $F_{\max} = F + W/2$. If W is sufficiently small compared to F, it is often advantageous to express such signals in *complex envelope* notation [Knight 1981]. The real-valued signal $x_c(t)$ is expressed in terms of a complex envelope $\tilde{x}_c(t)$:

$$x_c(t) = \sqrt{2}\,\Re\left(\tilde{x}_c(t)\,\mathrm{e}^{2\pi \mathrm{i} F t}\right). \tag{4}$$

The inverse transformation is (ideally)

$$\tilde{x}_c(t) = \sqrt{2}\,\left[x_c(t)\,\mathrm{e}^{-2\pi \mathrm{i} F t}\right]_{\mathrm{LP}}, \tag{5}$$

where $\Re()$ denotes the real part of a complex number, and the subscript LP denotes application of an ideal low-pass filter with a passband of $-W/2$ to $W/2$ and unity gain. The complex envelope $\tilde{x}_c(t)$ is advantageous because it varies much more slowly with time than does the real signal $x_c(t)$ (to the extent that $W \ll F$). The complex envelope may be sampled and interpolated using Eq. (3) just like the original real signal, but now the maximum sample interval h required to fully recover the signal is determined by the bandwidth, not the maximum frequency $F_{\max}$:

$$h \leq 1/W, \tag{6}$$

i.e., the Nyquist band extends from $F - 1/(2h)$ to $F + 1/(2h)$. As a general rule, the complex envelope notation is advantageous whenever the ratio of bandwidth to maximum frequency is less than about 25%.

Normalization: The constant $\sqrt{2}$ in Eqs. (4) and (5) is chosen to preserve power: $\mathbf{Av}\{|\tilde{x}_c(t)|^2\} = \mathbf{Av}\{[x_c(t)]^2\}$, where $\mathbf{Av}\{\}$ denotes a time average.

SST Class: To specify the complex envelope representation in any class derived from the base class **Signal**, set *frequency* to the center frequency F, *isComplex* to *true*, and *times* to a **UniformGrid** specifying the sample times.

SST Class: Transformations between real samples and complex envelope samples are implemented in SST class **ResampleSignal**, which uses approximate forms of Eqs. (4) and (5) employing window functions to combine finite-length filters with finite-order interpolation.

4.1.3 Windowed Frequency Domain

The third signal representation used in SST is a *windowed frequency domain* form. Currently, the primary application for the windowed frequency domain representation is in the implementation of class **DirectSpectrum**, which will be discussed in Sec. 8.

Conceptually, we break the continuous signal into blocks of a convenient size (one per *update cycle*) and perform a Fourier transform on each block. To avoid end effects, we enlarge the blocks so that they overlap, and multiply each one by a smooth "pre-window function" $w(t)$ whose properties are specified below. The transformation starting from a continuous signal is

$$X_c(f, t_u) = \int_{-\infty}^{\infty} w(t - t_u)\, x_c(t)\, \mathrm{e}^{-2\pi \mathrm{i} f(t-t_u)}\, \mathrm{d}t, \tag{7}$$

where the index u labels an update time on a uniform grid with interval Δ:

$$t_u = t_0 + u\Delta. \tag{8}$$

Note that the update interval Δ is normally much larger than the signal sampling frequency h used in Eq. (1).

The inverse operation, from windowed frequency domain to continuous signal, involves an inverse Fourier transform, multiplication by a "post-window function" $w'(t)$, a time shift, and a summation:

$$x_c(t) = \sum_u w'(t - t_u) \int_{-\infty}^{\infty} X_c(f, t_u)\, \mathrm{e}^{2\pi \mathrm{i} f(t-t_u)}\, \mathrm{d}f. \tag{9}$$

Of course, in the computer $X_c(f, t_u)$ is sampled on uniform grids in two dimensions, frequency f and update time t_u. For real signals the values of $X_c(f, t_u)$ for negative frequencies are not stored because

$$X_c(f, t_u) = X_c^*(-f, t_u), \tag{10}$$

where the asterisk represents complex conjugation.

The requirement that Eqs. (7) and (9) are inverses of one another imposes the following requirement on the two window functions:

$$\sum_u w'(t - t_u)\, w(t - t_u) = 1 \tag{11}$$

for all times t. Typically, the pre-window $w(t)$ is a Hann (cosine squared) window, and the post-window $w'(t)$ is rectangular. We will return to window functions at the end of this section.

Normalization: The sampled version of $X_c(f, t_u)$ is normalized such that it converges to Eq. (7) as the sampling interval $h \to 0$.

SST Classes: To specify a signal in windowed frequency domain form, use a subclass of class **Spectrum** with attributes *isPower* = *false*, *isComplex* = *true*, and *isCorrelated* = *false*. Attributes *times* and *frequencies* are **Grid** objects (Sec. 4.8) specifying the update times t_u and the frequencies f where the spectrum is sampled.

The transformations defined in Eqs. (7) and (9) are implemented by SST classes **SpectrumFromSignal** and **SignalFromSpectrum**, respectively.

4.2 Second Moment Time Series: Power Spectra and Scattering Functions

Each of the three representations in the previous subsection is "complete" in that it carries enough information to uniquely specify a continuous signal. SST also supports two types of time series, *power spectra* and *scattering functions*, that represent second-order statistical descriptors of a signal. These are "incomplete" in that they do not support unique reconstruction of the original signal.

4.2.1 Power Spectra

The *power spectral density* (PSD) of a stationary random signal is normally defined as the expectation of the Fourier transform of the autocovariance of the signal. For multi-channel signals, this definition is easily generalized to include covariances between channels. For nonstationary signals or those with non-random components (e.g., tones), the definition must be modified further to bring in window functions to ensure that the result is finite and to restrict it to a time interval over which the spectrum can be considered quasi-stationary.

A useful estimator of the PSD for multi-channel, non-stationary signals should be defined and normalized in such a way that it approximates the true PSD in cases where the signal is random and stationary. The expectation value of such an estimator is

$$P_{cc'}(f,t_u) = \frac{\mathbf{E}\left\{X_c(f,t_u)\, X^*_{c'}(f,t_u)\right\}}{\int_{-\infty}^{\infty} |w(\tau)|^2 \,\mathrm{d}\tau}, \tag{12}$$

where $X_c(f,t_u)$ is given by Eq. (7), $w(\tau)$ is the window function used in that equation, and $\mathbf{E}\{\cdot\}$ denotes the statistical expectation value. For signals in μPa, the units of $P_{cc'}(f,t_u)$ are $\mu\mathrm{Pa}^2/\mathrm{Hz}$. The diagonal elements ($c = c'$) are measures of the power per unit frequency in a given channel in the neighborhood of the update time t_u, where the neighborhood is defined by the weighting function $w(\tau)$. The off-diagonal elements ($c \neq c'$) are measures of the cross-correlation between channels versus frequency in the same neighborhood.

For stationary signals, Eq. (12) gives the true PSD convolved with the PSD of the window function $w(\tau)$. Hence the interpretation of Eq. (12) as a "time dependent PSD" is most meaningful if the time series is random and a window function $w(\tau)$ can be found that satisfies both of the following conditions:

- The time series $x_c(t)$ is quasi-stationary over the time interval where the window function $w(\tau)$ is significantly nonzero.
- The power spectrum $P_{cc'}(f,t_u)$ is quasi-stationary in frequency f over a frequency interval whose width is that of the interval in which the Fourier transform of the window function is significantly nonzero.

The first criterion favors short windows (in the time domain), whereas the second one favors long windows. As a practical matter, Eq. (12) is a "good" PSD estimate if the window length lies within a range of window lengths over which the result doesn't depend strongly on the window length.

SST Classes: SST class **Spectrum** is the base of a hierarchy from which all spectra derive — both complex vector-valued amplitude spectra $X_c(f,t_u)$ and matrix-valued power spectral densities $P_{cc'}(f,t_u)$. Their primary common feature is that they represent functions of both time and frequency. The two types of spectra are distinguished by a Boolean attribute *isPower*. More details about this family of classes will be given in subsections 4.3 and 4.4 below.

SST Classes: Equation (12) (omitting the expectation value operator) is implemented by SST class **SpectrumFromSignal** with attribute *isPower* set to *true*. (With *isPower* set to *false*, that class implements Eq. (7).)

4.2.2 Scattering Functions

A *scattering function* $Z_{rr's}(\Gamma, f, T)$ can be thought of as a generalization of the *intensity impulse response function* [Dahl 2001] for reverberation — generalized in that it includes Doppler spread, frequency dependence, and inter-channel correlation as well as time spread. It expresses the relationship between the PSD of a source signal and the PSD of the resulting reverberation signal received by a multi-channel sonar receiver. The details of that relationship are the subject of Sec. 10. For now, just consider it a matrix-valued function of Doppler shift Γ, frequency f, and two-way travel time T, for a given source channel s, where the function value is a non-negative definite square matrix (indices rr') whose dimension is the number of receiver channels. The Doppler shift Γ is defined as the ratio of received frequency to transmit frequency, which can differ from unity due to motion of the source, receiver, and scatterers. The scattering function is introduced here because it shares the data-flow design of signals and spectra, as outlined in the following subsection.

SST Classes: SST class **ScatFun** is the base of a hierarchy from which all scattering functions derive. All of them represent matrix-valued functions of Doppler, frequency, and time. More details about this family of classes will be given in the next two subsections and in section 10.

4.3 Data Flow Design

All of the SST classes that produce, modify, and store time series are based on a *data flow* design. They represent continuous functions of time, sampled on a uniform grid of time values. Each data-flow class is based on one of three base classes: **Signal**, which depends only on time, **Spectrum**, which depends on both frequency and time, and **ScatFun** (a scattering function), which depends on Doppler, frequency, and time. Subclasses within each hierarchy differ according to how samples are stored or computed. From the common "data flow" point of view, all such classes produce and/or consume a block of numbers (samples) for each value of time in some uniformly increasing sequence of times.

The samples themselves, however, may or may not be contained within an object of one of these classes. Instead, these objects should be regarded as sources or sinks of samples, which can be made to produce or accept blocks of samples, in time order, in response to *readBlock* or *writeBlock* requests on the object. Creating an object or assigning it to a variable merely establishes a link, and perhaps a path along which the samples will flow.

Most of SST's work is done under control of a **CopySignal** command, which simply reads successive blocks of data from one data-flow object and writes the resulting data into another such object. The object on the "write" side of a **CopySignal** command is typically rather simple: It just stores the data in a file or memory buffer in some specified format. The processing on the "read" side of a **CopySignal** command can be much more complex. Data-flow objects are typically organized into a network, in which each object might contain references to other data-flow objects that supply its inputs. For example, an object of class **VarDelay** contains references to at least two other data-flow objects: one to supply the signal to be delayed, and one to supply the time-varying delay by which the first signal is to be delayed.

The protocol is common to all such objects: In response to a "read" request, each object figures out which data it needs from each of its input data-flow objects, and issues corresponding "read" requests. These requests propagate upstream until they reach a "leaf" object that can satisfy its request, for example by computing its output or reading from a file. The resulting data propagate downstream as the requests are satisfied, as each object uses data from its input data-flow objects (if any) to compute its output. Eventually the block originally requested by the **CopySignal** command is stored by the object on the "write" side of the command, and the cycle repeats for the next block.

The following very simple SST script illustrates the concept:

```
# Variable Delay Example
insig = HarmonicFamily{
    isComplex = false
    times = UniformGrid:{ first=-2; last=5; rate=8000 }
    fundamental = 220  #Hz
    harmonics = (
      # number ampDB phaseDeg
        1       -3      0.0
        2       -6     90.0
    )
}
delay = InternalSignal{
    isComplex = false
    times = UniformGrid:{ first=0; last=5; interval=1 }
    buf = ( 1.00 1.10 1.25 1.45 1.7 2.0 )
}
outsig = SoundSignal{ file = "myDelayedSignal.snd" }
delayGenerator = VarDelay {
```

```
    isComplex = false
    times = UniformGrid:{ first=0; last=5; rate=8000 }
    inSignal = insig
    commonDelayBuf = delay
}
CopySignal delayGenerator outsig
```

This fragment creates four objects of various subclasses of **Signal**, and assigns them the user-selected names *insig*, *delay*, *outsig*, and *delayGenerator*. The **VarDelay** requires two input **Signal** objects as attributes: *inSignal* specifies the signal to be delayed, and *commonDelayBuf* specifies a time-dependent delay to be applied to all channels. In this case, the signal to be delayed consists of a **HarmonicFamily** specifying two tones sampled at 8 kHz, and the delay is an **InternalSignal** specifying a steadily increasing delay, sampled once per second.

The main point of this example is that, until the final **CopySignal** statement, the only object that contains samples of a signal is the **InternalSignal** called *delay*. The other objects merely specify how signals are to be produced or stored. The **CopySignal** statement works essentially like this: First **CopySignal** calls *openRead* on the input signal and *openWrite* on the output signal to get things started. Then it executes a loop that reads a block from *delayGenerator*, writes the result to *outsig*, and repeats. At the end of the input signal, it calls the *close* method on both input and output signals to shut them down. We have omitted complications related to starting, stopping, and managing buffers, but the central loop of **CopySignal** reduces to *readBlock*, *writeBlock*, repeat. Each subclass of **Signal** implements these operations in its own way:

- **VarDelay**::*readBlock* reads the necessary samples from each of the inputs (the delay and the signal to be delayed), interpolates the delay and the input signal to implement the delay, and returns the results. The block requested from the input signal is earlier than the output times, depending on the delay.

- **HarmonicFamily**::*readBlock* computes the requested samples and returns them to the caller.

- **InternalSignal**::*readBlock* returns the requested samples from its internal buffer.

- **SoundSignal**::*writeBlock* writes the samples into a file in a form that most computers can play through their speakers.

Requests for data propagate up the chain of *readBlock* operations and the results propagate back down, with a transformation at each step. The result, in this example, is an audio file with tones that get lower as the rate of change of delay gets larger.

The pattern illustrated by this example is typical of SST simulations, which have the following characteristics:

- Data-flow classes belong to any of three hierarchies, based at classes **Signal**, **Spectrum**, or **ScatFun**. All of them represent continuous functions of time. The three hierarchies differ in whether they depend on frequency or Doppler as well as time. Subclasses within each hierarchy differ according to how samples are stored or computed.

- Data-flow objects can represent nearly any time-dependent quantity, including some that are not normally considered "signals". These include the time-varying delay used to control a **VarDelay** object (as shown in the example above) and the time-varying filter coefficients used by **VarFirFilter**.

- Classes that compute samples on demand are *read-only*; they implement *readBlock* and *openRead*, but not *writeBlock* or *openWrite*. Many of the read-only objects accept other data-flow objects as attributes; these supply streams of input samples to be transformed. Chains of transformations formed in this way can be of any length. Read-only objects can be used as the source of a **CopySignal** command or as an input to a transformation.

- Classes that contain or store samples are *read-write*; they implement both *readBlock* and *writeBlock* operations, plus the *openRead* and *openWrite* methods. They can be used in the same contexts as the read-only objects, and can also be specified as the destination of a **CopySignal** command.

- SST's data-flow processing chains are *demand driven*: the caller determines which samples are required, and the called function delivers those samples. This "pull" model is in contrast to the "push" model used in most real-time signal processing systems, which are *input driven*; they process input data as they arrive.

- The class hierarchies satisfy the Liskov Substitutability Principle [Stroustrup 2000] (mostly): if a role (e.g., an attribute) calls for an object of one of the three base classes, any member of the same hierarchy can be used there. The main exception is the obvious one: read-only classes cannot be used as the destination of **CopySignal**.

4.4 Data Flow Classes

The classes shown in Tables 1, 2, and 3 represent various kinds of time series and are based on data-flow design.

Table 1: Signal Classes

Class	*Inputs*	*Summary*
Signal		Base class (abstract): multi-channel function of time
BareAsciiSignal	file	Signal in a simple ASCII formatted file
BinarySignal	file	Signal in a headerless encoded binary file
BroadbandNoise		Gaussian noise with given power spectrum
DirectSignal	Signal, Eigenrays	Sound propagated from source to receiver
FIRCoefBuf	Spectrum	Coefficients for variable FIR filter
FrequencyShiftSignal	Signal	Frequency-shifted copy of its input Signal
HarmonicFamily		Sum of harmonic tones
InternalSignal	memory	Signal in an internal buffer
LPFirCoefBuf		Coefficients for low-pass FIR filter
MergeSignal	Any DFs	Merges input flows into adjacent channels
ModulatedTone		Tone with frequency and/or amplitude modulation
ResampleSignal	Signal	Changes *isComplex*, *frequency*, *times*
ReverbSignal	ScatFun, Signal	Generate reverberation sound
SIOSignal	file	Signal in a binary SIO file (times etc. in header)
ScaleSignal	Any DF	Input flow times constant scale factor
SelectChannel	Any DF	One channel from a data flow
SignalFromSpectrum	Spectrum	Compute time series from amplitude spectrum
SoundSignal	file	Signal in .snd file format, for listening
SumSignal	Any DFs	Sum of other data flows
TargetEcho	Signal	Echo of active pulse from target
VarDelay	2-3 Signals	Delay input signal by time-varying amount
VarFirFilter	2 Signals	Variable finite impulse response (FIR) filter

In each table the *Inputs* column gives the types of any stream-like inputs; other inputs are omitted. Eigenrays are considered stream-like because they generate internal streams of eigenray properties (loss spectra, delays, and directions), as discussed in Sec. 8.

Table 2: Spectrum Classes

Class	*Inputs*	*Summary*
Spectrum	(abstract)	Base class: multi-channel function of frequency and time
AsciiSpectrum	file	Spectrum in an ASCII text file
BinarySpectrum	file	Spectrum in a plain binary file
DirectSpectrum	Spectrum, Eigenrays	Spectrum of sound propagated from source to receiver
FactorSpectrum	Spectrum	Cholesky factorization (matrix square root) of a power spectrum
GaussianSpectrum	Spectrum	Generate Gaussian random realization of a spectrum
InternalSpectrum	memory	Spectrum in an internal buffer
ReverbSpectrum	ScatFun, Spectrum	Convolve reverberation ScatFun with pulse spectrum
SIOSpectrum	file	Spectrum in a binary SIO file (times, frequencies, etc. in header)
SpectrumFromSignal	Signal	Analyze a Signal into a time-dependent Spectrum
UnfactorSpectrum	Spectrum	Square a factored Spectrum to get back a power spectrum
VarSpectFilter	Spectrum, Signal	Frequency Domain Finite Impulse Response filter

Table 3: Scattering Function Classes

Class	*Inputs*	*Summary*
ScatFun	(abstract)	Base class: multi-channel function of Doppler, frequency, and time
AsciiScatFun	file	A ScatFun in a simple ASCII formatted file
BBBDirectionalScat	Eigenrays	Broadband Bistatic Scattering Function using spherical tesselation
BBBScatFun	Eigenrays	Compute a Broadband Bistatic Scattering Function
InternalScatFun	memory	A ScatFun in an internal buffer
SIOScatFun	file	A ScatFun in a binary SIO file

Classes with "file" or "memory" in the *Inputs* column are read-write, and the rest are read-only. The "file" or "memory" referred to in that column can be used for input, output, or both. The samples in the "memory" classes can be used as input by setting the values from the command language as in the example in Sec. 4.3; they can be used as output by printing the object using SST's **print** command; or they can serve as temporary storage, written by one **CopySignal** command and read by another. The "file" classes can be uses similarly, except the samples are in a separate file.

Classes with "Any DF" or "Any DFs" in the *Inputs* column can accept as inputs classes from any of the three data flow hierarchies **Signal**, **Spectrum**, or **ScatFun**. They are "chameleons" that take on the logical character of their inputs. For example, a **SumSignal** object that has **ScatFun** objects as its inputs effectively becomes a **ScatFun**, and can be copied into an output **ScatFun** subclass such as **SIOScatFun**. The extra attributes that come with **ScatFun** (e.g. the *dopplers* grid) tunnel through the **SumSignal** from the inputs to the result. Unfortunately, this masquerade is incomplete: The chameleon classes can be used as inputs to other chameleon classes or the **CopySignal** command (which has a similar chameleon character), and they can be used anywhere that a **Signal** is required, but they cannot be used in other contexts where a **Spectrum** or **ScatFun** is required.

A few of the classes in Tables 1, 2, and 3 are high-level classes that are essential parts of the user's view of SST. These include **BBBScatFun**, **DirectSignal**, **ReverbSignal**, **SumSignal**, and **TargetEcho**; each of these classes will be discussed in more detail in subsequent sections. The rest are storage options, classes used internally by higher-level classes, and utilities that SST users have found useful. All of them are available for use by SST users through the command language, and all of them can be substituted anywhere that an object of the base class is required (subject to read/write constraints).

4.5 Basic Signal Operations

Signal processing operations are handled by those classes in Tables 1 and 2 that accept only other signals or spectra as inputs. Some of these have already been described, and others are simple and obvious. That leaves a few operations that are central to SST's operation: delays, filters, and noise generation.

4.5.1 Variable Delays

Ideally, SST class **VarDelay** accomplishes the following:

$$y_c(t) = x_c(t - T_c(t)), \tag{13}$$

where $x_c(t)$ is the input signal to be delayed (attribute *inSignal*). The time-varying delay $T_c(t)$ is the sum of two input **Signal** objects: a single-channel *commonDelayBuf* to be applied to all channels, and a multi-channel *channelDelayBuf* to be applied separately to corresponding channels of the signal. Either of the two delays may be omitted. Typically, the delays vary much more slowly than the signal itself, so the sampling rate for the delays is much lower than that of the input signal (typically a few samples per second or less). For example, in one application $T_c(t)$ is the sound propagation delay along one ray path from the source to the receiver, which changes with time because the source and receiver are moving.

The algorithm involves two interpolations. First, the delay $T_c(t)$ is interpolated to the sample times required for the output signal $y_c(t)$. Because $T_c(t)$ varies slowly, this first interpolation is always linear. Second, the input signal $x_c(t - T_c(t))$ is interpolated to the times given by the output sample times minus the interpolated delay. Typically this second interpolation must be done using a relatively high order; i.e., one must make use of a relatively large number of samples of the input signal in the neighborhood of each desired time $t - T_c(t)$.

The ideal band-limited interpolation formula, Eq. (3), has the disadvantage that it has a very long decay time — all of the samples in the signal are required to compute a single interpolated value. The first of two compromises used by class **VarDelay** is to introduce a window function to limit the number of samples used:

$$y_c(t) = \sum_{k=-M/2}^{M/2-1} d\left(\frac{t - t_{n'}}{h} + k\right) x_c(t_{n'} - kh), \tag{14}$$

where $M-1$ is the interpolation order (determined by the *order* attribute of the input signal $x_c(t)$) and $t_{n'}$ is the largest input sample time less than the desired time:

$$n' = \lfloor (t - t_0)/h \rfloor. \tag{15}$$

The interpolation weights $d(z)$ are the ideal weights from Eq. (3) multiplied by a Hann window of length M samples:

$$d(z) = \cos^2(\pi z/M) \left(\frac{\sin(\pi z)}{\pi z}\right), \tag{16}$$

where $-M/2 \le z \le M/2$. When the order is 3 or less ($M \le 4$), polynomial interpolation is used instead of Eq. (16).

The second compromise (if $M > 4$) is to tabulate the interpolation coefficients to avoid re-calculating them, using a time grid that is L times finer than the grid used for the signal:

$$d(z) \approx d\left(\left\lfloor \frac{z}{L} + \frac{1}{2} \right\rfloor L\right). \tag{17}$$

The sub-sampling ratio L can be specified by the user, but the default value of 512 is almost always sufficient.

To determine what interpolation order to use, it is helpful to view the interpolation algorithm as a band-pass filter. The spectrum of a discrete-time sampled signal consists of repeated copies of the desired spectrum, one within the Nyquist band (Eq. (2) or (6)) and others above and below the Nyquist band [Oppenheim Schafer 1989]. Interpolating it to a band-limited continuous signal is equivalent to filtering out all of the extra copies outside the Nyquist band.

In the ideal version of Eq. (3), the filter has a spectral response of unity throughout the Nyquist band, and zero elsewhere. Using a window function to reduce the filter length makes this transition more gradual. The width of this transition zone, for an M-sample Hann window, is roughly $2.5/(Mh)$ (from table 7.2 of [Oppenheim Schafer 1989]). Our objective is for the filter to have a spectral response near unity over the part of the Nyquist band in which the signal has significant power. In addition, we want the response to decrease to essentially zero at the locations of those extra copies, outside the Nyquist band, that must be filtered out. In between, near the edge of the band, a finite-length filter produces an incorrect result — so we must ensure that there are clear zones near the band edges where the input signal does not have significant power.

The suggested rule of thumb: if the significant parts of the signal cover a fraction α of the Nyquist band (in the range $F \pm \alpha/(2h)$ for complex signals, or 0 to $\alpha/(2h)$ for real signals), use an interpolation order $M - 1$ satisfying

$$M \ge 2.5/(1 - \alpha). \tag{18}$$

For example, for the common case of 80% band coverage ($\alpha = 0.8$), at least $M = 13$ is required. Even values of M (odd values of the **Signal** attribute *order*) are slightly more efficient, so *order* should be 13 or more.

The default value of *order* is 5, which is appropriate for no more than 50% band coverage.

4.5.2 Variable Finite Impulse Response Filters

SST needs filters whose spectral response depends (slowly) on time as well as frequency:

$$y_c(t) = \int_{-\infty}^{\infty} h_c(\tau, t)\, x_c(t-\tau)\, \mathrm{d}\tau, \tag{19}$$

where the filter impulse function $h_c(\tau, t)$ should ideally satisfy

$$h_c(\tau, t) = \int_{-\infty}^{\infty} H_c(f, t)\, \mathrm{e}^{2\pi \mathrm{i} f \tau}\, \mathrm{d}f, \tag{20}$$

where $H_c(f, t)$ is the filter's specified time-dependent spectral response. This treatment differs from the usual textbook case in that both $H_c(f, t)$ and $h_c(\tau, t)$ depend parametrically on time t. As written, this is theoretically sloppy (what does $H_c(f, t)$ really mean?), but as a practical matter it can be rescued by requiring that the change in $H_c(f, t)$ is negligible over intervals of t comparable to the range of τ [the width of the impulse response $h_c(\tau, t)$]. Equivalently, we require that the frequency response $H_c(f, t)$ varies slowly in frequency on a scale given by the inverse of the scale of its time variation. For example, in one application $H_c(f, t)$ is the sensitivity of a beam pattern to broadband sound arriving along one ray path from the source to the receiver, which changes with time because the source and receiver are maneuvering.

To eliminate the infinities in the integration limits (making a *finite impulse response*, or FIR, filter), we use the *window method* [Oppenheim Schafer 1989]: Eq. (20) is replaced by

$$h_c(\tau, t) = w(\tau) \int_{F-1/(2h)}^{F+1/(2h)} H_c(f, t)\, \mathrm{e}^{2\pi \mathrm{i} f \tau}\, \mathrm{d}f\,, \tag{21}$$

where $w(\tau)$ is a smooth window function whose length is short compared to the scale on which $H_c(f, t)$ varies with time t, and whose Fourier transform is narrow on the scale on which $H_c(f, t)$ varies with frequency f.

The choice of $w(\tau)$ enforces the smoothness requirements of the previous paragraph. The actual frequency response of this filter is the convolution of the desired frequency response, $H_c(f, t)$, with the Fourier transform of the window, $W(f)$. Hence, $W(f)$ should have a narrow central peak and low sidelobes. This issue is discussed at length in signal processing texts [Oppenheim Schafer 1989]. SST uses a Hann (cosine squared) window.

SST Class: Class **VarFirFilter** implements the discrete version of Eq. (19), given an input signal $x_c(t)$ and a stream of filter coefficients $h_c(\tau, t)$ sampled in both τ and t. Normally the sample interval in t for the coefficients is much larger than the sample interval common to τ and the signal. For a given lag τ the coefficients are

interpolated linearly in t between the input samples. The convolution is done directly, in the time domain, if the filter is short. For longer filters the convolution is done using fast Fourier transforms (FFTs). The break-even point, determined empirically, is currently at 48 samples in τ; the user can adjust it via a global parameter named *firFourierCutoff*.

SST Class: Class **FIRCoefBuf** is a data-flow class whose input stream is a time-dependent spectrum $H_c(f,t)$ giving the desired filter response, and whose output is a stream of filter coefficients $h_c(\tau,t)$. Most of its work is done by class **FIRCoef**. Together, these classes implement Eq. (21).

SST Class: Class **VarSpectFilter** is equivalent to **VarFirFilter**, except that its input and output signals are in the *windowed frequency domain* representation (Eq. (7)). Because **FIRCoefBuf** uses FFTs whenever it is faster than the time domain implementation, **VarSpectFilter** is advantageous only if the input is already in the frequency domain, or if the desired output is in the frequency domain, or if several filters are to be applied consecutively and the filters are relatively long. **VarSpectFilter** is used internally to implement class **DirectSpectrum**, which is discussed in Sec. 8.

4.6 Generating Signals

Most of SST is about what happens to signals between a source and a receiver. But where do the original signals come from? One option is from "outside": SST can read signals, spectra, and scattering functions in all of the same file formats that it uses for writing them (the entries with "file" or "memory" under the "Inputs" column in Tables 1, 2, and 3). If measured data or externally generated signals are available, simply put it in one of those forms to use it as an input to SST. If the external signal isn't quite right, use SST's signal processing tools (or external tool collections like Matlab) to filter, sum, delay, or resample them.

Most SST simulations, however, have no need of external input signals because SST provides a useful collection of simple tools to generate signals having specified properties. The remainder of this section outlines those tools.

4.6.1 Generating Gaussian Noise

Gaussian noise with a specified power spectrum can be used as a component of the signal put into the water by a source like a submarine or ship. To do that, use it as the *signal* component of a **Source** (sec. 7). Such noise can also be used as "background" noise, including sonar self-noise and other distributed noise sources for which SST

has no explicit model. To do that, use **SumSignal** to add it to the output signal from the simulation.

Given a power spectral density (PSD) $P_{cc'}(f, t_u)$, our objective is to generate a multi-channel Gaussian random signal $x_c(t)$ such that applying Eqs. (7) and (12) returns a close approximation to the original PSD. Such a signal is a "realization" of the original PSD. To be more precise, if the expectation operator $\mathbf{E}\{\cdot\}$ in Eq. (12) is replaced by an average over independent realizations, that average should converge to the original PSD as the number of realizations increases. This is a well-defined and common problem for stationary, single-channel signals, and extending the usual methods to multiple channels is straightforward. *Stationary*, in this context, means the PSD $P_{cc'}(f, t)$ is independent of time t.

To extend it efficiently to a nonstationary PSD, an additional assumption is required: that an update interval Δ exists such that the time variation of $P_{cc'}(f, t_u)$ is slow on a scale of Δ and the frequency variation is slow on a scale of $1/\Delta$. Under those conditions, the following variant of the method of Mitchell and McPherson [Mitchell McPherson 1981] generates acceptable Gaussian realizations of $P_{cc'}(f, t_u)$.

The first step is to *factor* the PSD: A *generator* $G_{cc'}(f, t_u)$ is needed that satisfies

$$P_{cc'}(f, t_u) = \sum_{c''} G_{cc''}(f, t_u)\, G^*_{c'c''}(f, t_u). \tag{22}$$

This factorization always exists because, for any given frequency f and time t_u, the matrix $P_{cc'}(f, t_u)$ is non-negative definite (its eigenvalues are positive or zero). This factorization is not unique, and there are several good ways to compute $G_{cc'}(f, t_u)$; SST uses Cholesky factorization [Golub Van Loan 1996], which is fast, reasonably stable, and produces a triangular result.

The second step is to multiply the generator by a vector of independent, complex, unit-variance Gaussian random numbers $g_c(f, t_u)$

$$X_c(f, t_u) = \sum_{c'} G_{cc'}(f, t_u)\, g_{c'}(f, t_u), \tag{23}$$

where the random numbers satisfy

$$\mathbf{E}\{g_c(f, t_u) g^*_{c'}(f', t_{u'})\} = \delta_{cc'}\, \delta_{uu'}\, \delta(f - f'), \tag{24}$$

where $\delta_{cc'}$ is a Kronecker delta function and $\delta(x)$ is a Dirac delta function. Of course, in the discrete domain the Dirac delta is effectively replaced by the Kronecker delta.

The third step is the same as Eq. (9): Inverse Fourier transform, window, and add. However, for this application the requirement on the post-window function is

$$\sum_u |w'(t - t_u)|^2 = 1 \tag{25}$$

instead of Eq. (11). This is satisfied, for example, by a set of cosine windows with 50% overlap. However, the Mitchell-McPherson window function [Mitchell McPherson 1981], which also satisfies Eq. (25), has somewhat better spectral properties.

SST Classes: The Mitchell-McPherson method of generating nonstationary Gaussian noise is implemented by the sequence of **FactorSpectrum** (Eq. (22)), **GaussianSpectrum** (Eq. (23)), and **SignalFromSpectrum** (Eq. (9)). This same sequence is used within class **ReverbSignal** to generate realizations of reverberation. For stationary, single-channel noise, a simpler implementation is provided by class **BroadbandNoise**.

4.6.2 Generating Harmonic Tone Families

Harmonic tone families normally represent machinery noise. The usual way to generate the noise emitted by a submarine, ship, or weapon is to use **SumSignal** to combine several **HarmonicFamily** objects with a **BroadbandNoise** object. The result is used as the *signal* component of a **Source** object representing the vehicle.

The signal generated by a **HarmonicFamily** object has the following form:

$$x(t) = \sum_n A_n \cos(2\pi n f_1 t + \phi_n), \tag{26}$$

where the frequency of each term is an integer multiple of f_1. The user specifies f_1 (attribute *fundamental*) and a three-column table giving the harmonic number n, the amplitude in decibels ($20\log(A_n)$), and the phase at $t = 0$ in degrees ($(180/\pi)\phi_n$) for each harmonic.

4.6.3 Generating Modulated Tones

Class **ModulatedTone** is designed primarily to generate the pulses transmitted by the transmitter of an active sonar system. A **ModulatedTone** object, or several **ModulatedTone** objects combined using a **SumSignal**, may be used to generate almost any of the pulses commonly used by active sonar systems.

The signal generated by a **ModulatedTone** has the following form:

$$\begin{aligned} x(t) &= A\, e(t)\, \cos(\phi(t)) \\ \phi(t) &= \phi_0 + \int_{t_0}^{t} m(\tau)\, \mathrm{d}\tau \\ e(t) &= \mathit{envelope}(t) \\ m(\tau) &= 2\pi\, \mathit{frequencyModulation}(\tau) \\ A &= 10^{\mathit{level}/20} \\ \phi_0 &= (\pi/180)\, \mathit{startingPhase} \end{aligned} \tag{27}$$

where *envelope*, *frequencyModulation*, *level*, and *startingPhase* are user-specified attributes of class **ModulatedTone**. In particular, *envelope* and *frequencyModulation*, which specify the time-dependent amplitude and frequency of the generated tone, are *function objects* (objects of any subclass of base class **Function**) that can represent any arbitrary function of time. Often objects of class **TableFunction** are used here. The window functions described in Sec. 4.7 are often useful for the *envelope* attribute. If *envelope* is omitted, it is a constant function with value 1.0. If *frequencyModulation* is omitted for complex envelope signals, it is a constant function whose value is the signal's center frequency (F in Eq. (4)).

The integration used to compute the phase $\phi(t)$ in Eq. (27) is done numerically using two-point (third order) Gaussian integration from each output sample to the next. This is exact for polynomial frequency modulation functions up to cubic. The phase is always continuous throughout the pulse sequence.

The SST Web contains examples showing how to use **ModulatedTone** to specify shaded pure-tone pulses, FM sweeps, and sequences of shaded or unshaded tones or sweeps. When used with **SumSignal**, **ModulatedTone** can also generate chords and other non-sinusoidal signals.

4.7 Window Functions

Table 4 shows the window functions that SST provides for use with **SpectrumFromSignal**, **SignalFromSpectrum**, or **ModulatedTone**. Those marked "Squared, Eq. (25)" in the "Sum Rule" column are appropriate for generating noise or reverberation. Those listing "Linear, Eq. (11)" can be used as either the pre-window $w(t)$ or the post-window $w'(t)$ for transforming signals between time-domain and windowed frequency-domain representations (Eqs. (7) and (9)), provided the other window is a **RectangularWindow**. Window functions used for power spectrum analysis (Eq. (12)) or as the *envelope* of a shaded **ModulatedTone** need not satisfy any particular sum

rule; for those applications, **TaylorWindow** or **HannWindow** is often preferred to achieve a narrow spectral peak and low sidelobes. Because window functions are derived from base class **Function**, you can use **TableFunction** to enter any window you want. For a general discussion of window functions, refer to Oppenheim and Schafer [Oppenheim Schafer 1989].

Table 4: Window Function Classes

Class	*Sum Rule*	*Summary (see Eq. 28)*
Function		Base class: real function of one real argument
TableFunction		Function specified as table of value vs. argument
CosineWindow	Squared, Eq. (25)	$\cos(\pi x/2)$
HannWindow	Linear, Eq. (11)	$\cos^2(\pi x/2)$
LinearWindow	Linear, Eq. (11)	$1 - x$
MitchellMcPhersonWindow	Squared, Eq. (25)	Mitchell-McPherson [Mitchell McPherson 1981]
RectangularWindow		1.0 in window, else 0
TaylorWindow		Taylor (low sidelobes) [Taylor 1955]

In the equations in Table 4, the variable x is a normalized form of the independent variable:

$$\begin{aligned} x &= (|t - \bar{t}| - (t_1 - \bar{t}))/L + 1 \\ \bar{t} &= (t_1 + t_0)/2 \\ L &= F(t_1 - t_0) \end{aligned} \tag{28}$$

where t_0 is the lower limit of the window (attribute *start*, default -1), t_1 is the upper limit (attribute *end*, default +1), and F is the fraction of the window length over which it is tapered (attribute *taperFraction*, default and maximum 1/2). The equations in the table apply only to the tapered region, $0 \le x \le 1$. Each of the window functions has the value 0 if $x \ge 1$ and 1.0 if $x \le 0$.

4.8 Grids

In many places, SST represents continuous functions in terms of samples at discrete values of an independent variable. SST provides a uniform mechanism for specifying these independent values: class **Grid** and its subclasses. They are used, for example, to specify the *times* attribute in a **Signal**, the *times* and *frequencies* of a **Spectrum**, or

the *times*, *frequencies*, and *dopplers* of a **ScatFun**. They are also used to specify the independent variables of **TableFunction** or **TableFunction2** objects, which can be used to specify window functions, the power spectrum in **BroadbandNoise**, and environmental parameters such as sound speed versus depth, volume absorption versus frequency, or ocean depth versus location.

Class **Grid** is the base class for a family of classes designed to specify an ordered set of values of an independent variable where some function is to be evaluated, or sampled. **UniformGrid** is the simplest subclass of **Grid**, in which the interval between adjacent values is constant (Eq. (1)). Other members of the family include **GeometricGrid** (in which the ratio between adjacent values is constant), **ListGrid** (for arbitrary values), and **SubGrid** (a subset of values from some other **Grid**).

5 The Eigenray Model

The model that describes how sound is transformed as it propagates from one part of the ocean to another — the eigenray model — is central to SST's operation. This transformation requires two time-dependent sets of inputs: the sound to be transformed, and the set of losses and delays to be applied to that sound. The eigenray model supplies the second set in the form of a finite list of eigenrays, each of which represents a distinct path by which sound can travel from one place in the ocean to another. Each eigenray takes as its input the source and receiver locations in the ocean, $\boldsymbol{r}_S$ and $\boldsymbol{r}_R$. Given that pair, eigenray p produces the following information:

- $T_p(\boldsymbol{r}_R, \boldsymbol{r}_S)$ is the propagation delay for path p; i.e., the time required for sound to travel from one end of the path to the other.
- $L_p(f, \boldsymbol{r}_R, \boldsymbol{r}_S)$ is the complex-valued propagation loss factor for path p including spreading loss, boundary reflection loss, and volume absorption. Its unit is inverse length (m^{-1}). It gives the ratio of the sound pressure at $\boldsymbol{r}_R$ to the transmitted sound pressure at $\boldsymbol{r}_S$ reduced to a distance of one meter, as a function of acoustic frequency f.
- $\boldsymbol{S}_{Sp}(\boldsymbol{r}_R, \boldsymbol{r}_S) = \nabla_S T_p(\boldsymbol{r}_R, \boldsymbol{r}_S)$ is the *slowness vector* at the source, which is defined as the spatial gradient of the time delay with respect to the source location $\boldsymbol{r}_S$. Its direction is opposite the direction of propagation along the eigenray at the source, and its magnitude is the inverse of the local sound speed at the source.

- $\boldsymbol{S}_{Rp}(\boldsymbol{r}_R, \boldsymbol{r}_S) = \nabla_R T_p(\boldsymbol{r}_R, \boldsymbol{r}_S)$ is the slowness vector at the receiver R; i.e., the spatial gradient of the time delay with respect to the receiver location $\boldsymbol{r}_R$. It points in the direction of propagation along the eigenray at the receiver, and its magnitude is the inverse of the local sound speed at the receiver.

At one level, the eigenray model consists of a finite set of maps whose inputs are two end-point locations, $\boldsymbol{r}_R$ and $\boldsymbol{r}_S$, and whose outputs are the four pieces of information listed above: T_p, $L_p(f)$, $\boldsymbol{S}_{Sp}$, and $\boldsymbol{S}_{Rp}$.

At a slightly higher level, the presence of the gradients $\boldsymbol{S}_{Sp}$ and $\boldsymbol{S}_{Rp}$ suggests use of a first-order Taylor expansion to compute the time delay, not just between the given end points, but between two points in small neighborhoods around the given end points:

$$T_p(\boldsymbol{r}_R + \boldsymbol{\rho}_R, \boldsymbol{r}_S + \boldsymbol{\rho}_S) = T_p(\boldsymbol{r}_R, \boldsymbol{r}_S) + \boldsymbol{S}_{Rp} \cdot \boldsymbol{\rho}_R + \boldsymbol{S}_{Sp} \cdot \boldsymbol{\rho}_S + \cdots , \tag{29}$$

where the offset vectors $\boldsymbol{\rho}_R$ and $\boldsymbol{\rho}_S$ are assumed to be small compared to the total propagation distance. This is useful because sources and receivers have finite size. Given an eigenray from one point on the source to one point on the receiver, this expansion can be used to compute (approximately) the delay from any part of the source to any part of the receiver.

For most purposes the "local plane wave" approximation (first-order Taylor expansion) of Eq. (29) is sufficient. However, for a long array and a source at short range, wave-front curvature may be significant. More to the point, large-aperture passive sonars like towed arrays use wave-front curvature as an important clue for estimating target range. The eigenray model does not give enough information for a full second-order expansion of the time delay, but SST gets part way there using a *spherical wave* approximation. In this approximation, the wave front at each end is locally spherical, with a center located at the apparent location of the other end as inferred from the time delay and the local sound speed. Each of the dot products in Eq. (29) is replaced by a second-order expression of the form

$$\tau_p(\boldsymbol{S}, \boldsymbol{\rho}) = \boldsymbol{S} \cdot \boldsymbol{\rho} + \frac{|\boldsymbol{S}|^2 |\boldsymbol{\rho}|^2 - (\boldsymbol{S} \cdot \boldsymbol{\rho})^2}{2T_p}. \tag{30}$$

This approximation is good for straight-line propagation, and remains useful for horizontal curvature in most scenarios. It is less trustworthy for vertical curvature in the presence of ray bending.

Another important property is that the delay T_p does not depend on frequency because the ocean is *nondispersive*; i.e., the dependence of the sound speed on frequency is very weak [Urick 1983]. Small violations of the nondispersive assumption can be absorbed into the phase of the complex loss $L_p(f, \boldsymbol{r}_R, \boldsymbol{r}_S)$.

It is equally important to note that the eigenray model does not give gradients of the propagation loss $L_p(f, \boldsymbol{r}_R, \boldsymbol{r}_S)$. This is a value judgment, not a fundamental limitation. The eigenray model computes the gradients of time delay because beam formers are very sensitive to variation in propagation time between one part of a source or sonar and another part. However, variation in propagation loss in a region the size of a source or sonar is not so important, so it is ignored. The loss $L_p(f, \boldsymbol{r}_R + \boldsymbol{\rho}_R, \boldsymbol{r}_S + \boldsymbol{\rho}_S)$ is assumed to be independent of $\boldsymbol{\rho}_R$ and $\boldsymbol{\rho}_S$ over such regions. Large sonar arrays or targets for which these assumptions do not hold can still be modeled, but to do so SST must treat them as compact groups of elements or highlights, and explicitly compute the eigenrays to the center of each group.

Thus, at the second level, the inputs to the eigenray model are not two points, but two *regions* around those points. For each eigenray, the eigenray model gives the loss and delay from any point in one region to any point in the other region.

There is a third level, too: the eigenray model specifies a *linear transformation* whose input is a sound source field $q_S(t, \boldsymbol{r})$ emitted from various parts of a source in the neighborhood of $\boldsymbol{r}_S$, and whose output is the resulting sound field $p(t, \boldsymbol{r})$ in the neighborhood of $\boldsymbol{r}_R$. SST assumes that this transformation has the following mathematical form (in the time domain):

$$\begin{aligned} p_R(t, \boldsymbol{r}_R + \boldsymbol{\rho}_R) \;\; = \;\; & \sum_p \int \int l_p(\tau, \boldsymbol{r}_R, \boldsymbol{r}_S) \\ & \times q_S\left(t - T_p(\boldsymbol{r}_R + \boldsymbol{\rho}_R, \boldsymbol{r}_S + \boldsymbol{\rho}_S) - \tau, \boldsymbol{r}_S + \boldsymbol{\rho}_S\right) \mathrm{d}\tau \, \mathrm{d}\boldsymbol{\rho}_S. \end{aligned} \tag{31}$$

The offset vectors $\boldsymbol{\rho}_R$ and $\boldsymbol{\rho}_S$ have the same scale as the size of the receiver and source, respectively, and are assumed to be small compared to the total propagation distance. The sum is over eigenrays (paths) p.

The propagation impulse response $l_p(\tau, \boldsymbol{r}_R, \boldsymbol{r}_S)$ in Eq. (31) is the inverse Fourier transform of the propagation loss:

$$l_p(\tau, \boldsymbol{r}_R, \boldsymbol{r}_S) = \int_{-\infty}^{\infty} L_p(f, \boldsymbol{r}_R, \boldsymbol{r}_S) \, \mathrm{e}^{2\pi \mathrm{i} f \tau} \, \mathrm{d}f. \tag{32}$$

Because $L_p(f, \boldsymbol{r}_R, \boldsymbol{r}_S)$ varies slowly with frequency, $l_p(\tau, \boldsymbol{r}_R, \boldsymbol{r}_S)$ is sharply peaked in time delay τ. Eq. (31) will be combined with the source and receiver models in Sec. 8.

5.1 Straight-line Eigenray Model

Of SST's three choices for the eigenray model, the only one that is contained entirely within the SST software is the base class, **EigenrayModel**. This model is valid only

if the sound speed is independent of depth and location, the ocean depth is constant, and the surface and bottom properties are the same everywhere. Sound travels in straight lines except where it reflects in the specular (mirror) direction from the surface or bottom.

SST computes the eigenrays using the *method of images*. Effectively, the sound travels in a straight line from the source to an *image* of the receiver, which is placed at the apparent location of the receiver as seen from the source, possibly via surface and bottom reflections. Each image is above or below the actual position of the receiver, by a distance that depends on the ocean depth and the number and order of the reflections. The vector from the source to the image of the receiver for path p is given by

$$\boldsymbol{R}_p = \boldsymbol{r}_R - \boldsymbol{r}_S - 2|S_p - B_p|z_R\hat{z} \pm 2B_pD\hat{z} \tag{33}$$

where B_p is the number of bottom reflections for path p, S_p is the number of surface reflections, D is the ocean depth, $\boldsymbol{r}_R$ is the receiver location, $\boldsymbol{r}_S$ is the source location, $\hat{z}$ is a unit vector in the z direction (down), and $z_R = \boldsymbol{r}_R \cdot \hat{z}$ is the receiver depth. The sign of the last term is + if the reflection closest to the source is from the bottom, or − if it is from the surface. There are four images (i.e., four paths p) for each nonzero value of B_p: $S_p = B_p - 1$, $S_p = B_p + 1$, and two with $S_p = B_p$.

The time delay for path p is

$$T_p = R_p/c \tag{34}$$

where $R_p = |\boldsymbol{R}_p|$ is the slant range from the source to the image of the receiver and c is the sound speed.

The propagation loss includes spherical spreading, volume absorption, and reflection losses:

$$L_p(f) = R_p^{-1}\mathrm{e}^{\alpha(f)R_p}\left[L_S(s,f)\right]^{S_p}\left[L_B(s,f)\right]^{B_p} \tag{35}$$

where $L_S(s,f)$ is the reflection loss for each surface bounce, $L_B(s,f)$ is the reflection loss for each bottom bounce, s is the sine of the grazing angle, f is the frequency, and $\alpha(f) \le 0$ is the volume absorption rate per meter (from the volume attenuation model, Sec. 6.3).

The slowness vectors are given by:

$$\begin{aligned} \boldsymbol{S}_{Rp} &= \boldsymbol{R}_p/(cR_p) \\ \boldsymbol{S}_{Sp} &= -\boldsymbol{R}_p/(cR_p) \end{aligned} \tag{36}$$

except that the sign of the vertical (Z) component of $\boldsymbol{S}_{Rp}$ is reversed if the total number of bounces, $S_p + B_p$, is odd.

5.2 CASS/GRAB Eigenrays

The Comprehensive Acoustic System Simulation (CASS) [Weinberg et al. 2001] software, developed by Henry Weinberg and others at NUWC Newport, models oceans in which the sound speed may vary with all three dimensions, and the bathymetry and bottom type may depend on horizontal location. The eigenrays can bend, and the direction of the forward bottom reflection depends on the local bottom slope. CASS is a self-contained modeling environment that can compute and plot many different results of interest for sonar analysis and performance prediction. The only component of CASS used by SST is the Gaussian Ray Bundle (GRAB) [Weinberg Keenan 1996] eigenray model.

CASS produces eigenrays in external files, which SST reads and uses in its simulations. Two different SST classes may be used, depending on the relationship between the CASS run and the SST run. The recommended choice is to run only SST, and specify class **CASSEigenrayRun** for the eigenray model. In that case, SST generates the CASS input files based on its own input parameters, and runs CASS as a subprocess. This is the recommended choice in most cases because SST ensures that the CASS and SST run streams are consistent. It is also less work than **CASSEigenrayModel** because you need to prepare only SST's input files, not CASS's.

SST users may choose to prepare CASS input files, run CASS to generate eigenrays, and then run SST independently. In that case, choose class **CASSEigenrayModel** and specify the names of the eigenray files that CASS wrote. This is strongly discouraged because it is very difficult to maintain consistency between CASS and SST. **CASSEigenrayModel** remains in SST primarily to facilitate testing.

5.3 Generic Sonar Model (GSM) Eigenrays

The Generic Sonar Model (GSM) [Weinberg 1985], a much older product of Henry Weinberg at NUWC, models oceans in which the sound speed may depend only on depth. The sound speed, ocean depth, and boundary properties are assumed to be independent of horizontal location (*range independent*). GSM includes five different user-selectable eigenray models, of which the most useful with SST (in our opinion) are the Multipath Expansion (MULTIP) and Fast Multipath Expansion (FAME) models.

Use of GSM is somewhat simpler than use of CASS, but otherwise very similar. Like CASS, GSM can be run separately before SST, using the SST class **GSMEigenrayModel**. Your other choice is to run only SST, specifying class **GSMEigenrayRun**

for the eigenray model; SST then runs GSM as a subprocess. The same issues of consistency arise with GSM as with CASS, but because GSM is simpler, the probability of getting it right with **GSMEigenrayModel** is somewhat better than with **CASSEigenrayModel**. Nevertheless, **GSMEigenrayRun** is the recommended option.

NUWC no longer supports GSM. We continue to support its use in SST, at least until our users have time to convert their SST scripts to CASS.

5.4 Eigenray Interpolation and Ray Identity

For many reasons, it is important that the delay, loss, and slowness vectors for a given eigenray are continuous functions of the end points $\boldsymbol{r}_R$ and $\boldsymbol{r}_S$. Continuity of the delay T_p is especially important, in part because the first derivative of the delay is the Doppler shift, to which many sonar receivers are especially sensitive. Achieving continuity for the straight-line eigenray model is easy because its properties are computed as needed, and because the identity of each eigenray is uniquely specified by the number and order of its surface and bottom reflections.

For both CASS and GSM eigenrays, the eigenray files read by SST contain eigenrays (sets of eigenray properties) at discrete values of the end point locations. Values between those locations must be computed using interpolation. Given a list of eigenrays at one location and another such list at a neighboring location, we are faced with the problem of matching the members of one list with the members of the other list in such a way that it makes physical sense to interpolate eigenray properties between the matching eigenrays at different locations. Matching eigenrays are assigned to a single *ray* (an object of some subclass of base class **Ray**). The expectation is that for a given *ray* object identified by index p, the maps from the location pair $(\boldsymbol{r}_R, \boldsymbol{r}_S)$ to the properties T_p, $L_p(f)$, $\boldsymbol{S}_{Sp}$, and $\boldsymbol{S}_{Rp}$ are continuous and physically meaningful.

CASS and GSM help with this matching by providing a *signature* for each eigenray, consisting of the numbers of surface bounces, bottom bounces, upper vertexes, and lower vertexes for each one. Unfortunately, the signature is not sufficient to specify uniquely how to match eigenrays at different locations because there can be any number of eigenrays (including zero) with the same signature for the same location pair, and that number can change from location to location.

At the start of a simulation, SST reads the CASS or GSM eigenray files into its memory and organizes them into a set of internal tables designed to support interpolation in the tables. To do this, each set of eigenray properties read from the files is assigned a *ray identity*, and eigenrays with the same identity are eventually mapped into the same *ray* object. Eigenrays with the same signature are assigned their ray

identities using a complex set of heuristics that take into account the closeness of the attributes (especially delay) of one eigenray to a simple extrapolation of the attributes of an adjacent eigenray. These heuristics work well in our tests, but our tasks would be much easier and our confidence would be higher if the eigenray models provided a unique signature to support unambiguous matching for interpolation.

6 Ocean Model

The primary inputs to the eigenray model (besides the locations of the end points) come from properties of the *ocean*, which is an object of class **Ocean**. The properties used as inputs to the eigenray model are the depth, sound speed, volume attenuation, and the reflection coefficients of the surface and bottom. These properties are used directly by the straight-line **EigenrayModel**, and they are passed in table form to CASS or GSM by class **CASSEigenrayRun** or **GSMEigenrayRun**, respectively. Classes **CASSEigenrayModel** or **GSMEigenrayModel** do not use them, since they get their inputs from the user-supplied CASS input file.

In addition, class **Ocean** supplies the scattering strengths used by SST's reverberation model, which will be described in Sec. 10.

6.1 Ocean Depth

The *depth* attribute of class **Ocean** is, in general, a function of horizontal location $D(x, y)$, in meters, with x and y in meters north and east of an arbitrary origin. Its type is class **Function2**, which is an abstract base class for describing any real-valued function of two real variables. That means the user can assign to *depth* an object of any subclass of **Function2**. By default, it is an object of class **ConstantFunction2**; the user simply assigns a value, e.g., "`ocean.depth = 200`".

If CASS eigenrays are used, the bathymetry can be specified by assigning to the *depth* attribute an object of class **TableFunction2**, which provides several ways to enter a table of depth on a rectangular grid versus horizontal location (meters north and east of an arbitrary origin). Interpolation in the table is normally bilinear (order 1), although the user can specify the order up to cubic (order 3).

Alternatively, assign to **Ocean**'s *depthFile* attribute the name of a file in CASS's input format containing a "`BOTTOM DEPTH TABLE`" section. SST will search for that section, parse it, extract the bathymetry, and build a **TableFunction2** object from the data.

6.2 Ocean Sound Speed

The *soundspeed* attribute of class **Ocean** is, in general, a function of depth $c(z)$ in meters per second, versus depth z in meters. Its type is class **Function**, which is an abstract base class for describing any real-valued function of one real variables. By default, it is an object of class **ConstantFunction**; the user simply assigns a value, e.g., "`ocean.soundspeed = 1500`". The default value is 1520 m/s.

If CASS or GSM eigenrays are to be used, the sound speed profile can be specified by assigning to the *soundspeed* attribute an object of class **TableFunction**, which provides several different ways to enter a table of sound speed versus depth. This table is passed to CASS or GSM, which does its own interpolation.

CASS permits the sound speed to depend on horizontal location as well as depth. However, SST does not yet provide a way to specify this horizontal variability.

6.3 Ocean Volume Attenuation

The *volumeAttenuation* attribute of class **Ocean** is, in general, a function of frequency $A(f)$ in dB per km, versus frequency f in Hz. By default, it is an object of class **ConstantFunction**; the user simply assigns a value, e.g., "`ocean.volumeAttenuation = -2.1`". The default value is 0 dB/km.

To specify frequency-dependent attenuation, one option is to assign to the *volumeAttenuation* attribute an object of class **TableFunction**, which allows the user to enter a table of attenuation versus frequency using tables of the same formats used for the sound speed. Another option is to assign **ThorpAttenuation** to the attribute. This option computes the absorption using the expression in [Urick 1983] page 108.

6.4 Surface and Bottom Models

Class **Ocean** contains attributes *surface* and *bottom*, both of which are objects of class **Boundary**. SST users can assign to them any object of any class derived from **Boundary**, including **APLBottom**, **JacksonBottom**, **APLSurface**, **GilbertSurface**, and **McDanielSurface**.

6.4.1 Reflection Coefficients

Each of these classes defines two complex-valued member functions *TotalForwardAmp* and *CoherentForwardAmp*, each of which is a function of two arguments, *sinAngle* (the sine of the grazing angle) and *frequency* (in Hz).

The straight-line **EigenrayModel** class uses these functions to compute the factors $L_S(s, f)$ and $L_B(s, f)$ in the eigenray propagation loss, Eq. (35); classes **CASSEigenrayRun** and **GSMEigenrayRun** use them similarly. For each boundary, SST calls the member function *CoherentForwardAmp* when the eigenrays are being used for one-way propagation or target echoes, but it calls *TotalForwardAmp* when the eigenrays are being used for reverberation. The theory behind this practice is based on the following distinction:

- For reverberation, the distinction between near-specular forward scattering and specular reflection is unimportant; all that matters is how much energy is removed from the total. Hence, SST calls *TotalForwardAmp*, which treats forward-scattered energy as if it were specularly reflected.
- For passive reception or target echoes, scattered energy is effectively lost because scattering reduces the coherence of the signal (reducing processing gain), stretches it out in time, and spreads it in angle, all of which tend to push it down under the background. Hence, SST calls *CoherentForwardAmp*, which treats forward-scattered energy as if it were absorbed.

This theory is flawed; for many purposes *TotalForwardAmp* is too large and *CoherentForwardAmp* is too small. We are currently addressing this issue by developing new algorithms to control the coherence of the simulated signal; the "coherent versus incoherent" dichotomy will be replaced by explicit and quantitative control of coherence.

For most purposes the distinction is unimportant anyway because *CoherentForwardAmp* is used only by the straight-line **EigenrayModel**. If **CASSEigenrayRun** or **GSMEigenrayRun** is selected, the *TotalForwardAmp* function is called repeatedly with different values of frequency and grazing angle to build a "`SURFACE REFLECTION COEFFICIENT TABLE`" and a "`BOTTOM REFLECTION COEFFICIENT TABLE`" to be passed to CASS or GSM. This tends to overestimate the reflection coefficient, except for reverberation. The table is passed in decibels; CASS and GSM ignore the phase.

(*Note*: CASS accepts a table of phase shifts too, but **CASSEigenrayRun** doesn't provide it. This needs to be fixed soon.)

6.4.2 Bistatic Scattering Strength

Each **Boundary** subclass also provides a function called *BistaticStrength*, denoted $S_l(f, \boldsymbol{S}_p, \boldsymbol{S}_q)$. It computes the scattering differential cross section per unit area of the boundary; it is dimensionless (an area divided by an area). It takes as its input a frequency f and two slowness vectors, $\boldsymbol{S}_q$ giving the direction from which incident sound arrives at the boundary, and $\boldsymbol{S}_p$ giving the direction toward which it scatters. The subscript l denotes a scattering layer (surface or bottom) and the subscripts p and q denote the outgoing and incoming eigenrays.

Most of the **Boundary** subclasses are fully bistatic. That means the two slowness vectors reduce to at least three independent parameters: two *grazing angles* (from the boundary plane to each of the two slowness vectors) and a *bistatic angle* (from a vertical plane containing one vector to the other vector). Under most conditions, the bistatic strength is strongly peaked in the region of the specular direction.

The older **Boundary** subclasses, **APLBottom** and **APLSurface**, are monostatic. That means they define a backscattering strength $S_l(f, \sin(\theta))$, which is a function of only one angle, the grazing angle θ. For those classes, the *BistaticStrength* method computes the backscattering strength for each of the two grazing angles (incoming and outgoing) and returns the geometric average (the square root of the product). This is the same form used by CASS. It is a reasonably good approximation for geometries that are close to backscattering (i.e., if the two slowness vectors are nearly equal) and for bubble-dominated surface scattering. It is a particularly bad approximation for the region near the specular direction because it does not produce a peak in that region. The monostatic models may be removed in a future SST release.

6.4.3 Boundary Classes

The **Boundary** classes are listed in Table 5 with the literature references from which they were taken and a brief indication of applicability. In the table "high frequency" means over 10 kHz and "mid frequency" means 1 to 10 kHz (roughly).

The base class **Boundary** allows the user to enter either or both of the reflection coefficient functions and the monostatic backscattering strength as tables versus angle and frequency. By default the reflection coefficients are unity (no loss) and the backscattering strength is zero in energy terms (no scattering). **Boundary** may be used for either the surface or the bottom.

For the surface models **APLSurface**, **GilbertSurface**, and **McDanielSurface**, *TotalForwardAmp* includes all of the incident intensity except the fraction that is ab-

Table 5: Surface and Bottom Models

Class	*References*	*Use*
Boundary		Monostatic, table driven
APLBottom	[APL Models 1994, Mourad Jackson 1989]	Bottom, monostatic, high frequency
JacksonBottom	[Williams Jackson 1998, APL Models 1994, Mourad Jackson 1993, Mourad Dahl Jackson 1991, Moe Jackson 1994, Schulten Anderson Gordon 1979]	Bottom, bistatic, mid to high frequency
APLSurface	[APL Models 1994]	Surface, monostatic, high frequency
GilbertSurface	[Gilbert 1993, Kulbago 1994]	Surface, bistatic, low to mid frequency
McDanielSurface	[McDaniel 1990, Lang Culver 1992, Donelan Hamilton Hui 1985]	Surface, bistatic, high frequency

sorbed by bubbles and converted to heat. *CoherentForwardAmp* also removes scattered intensity using a simple model of surface roughness [APL Models 1994]; this loss can be very substantial whenever the wave height is comparable to or greater than the acoustic wavelength. The only environmental input to these models is the wind speed.

For the bottom models **APLBottom** and **JacksonBottom**, the *TotalForwardAmp* and *CoherentForwardAmp* are identical. The reflected intensity includes all of the incident intensity except the fraction that is refracted into the bottom, as estimated using a lossy Rayleigh coefficient [Mackenzie 1959]. Bottoms of these classes may be specified using sets of parameters describing surface roughness, sediment sound speed, absorption rate, and scattering within the sediment. Sets of named bottom types (e.g., *MediumSand* or *CoarseSilt*) are also provided.

6.5 Volume Scattering Strength

Volume scattering tends to occur in *layers* because sea life tends to congregate at restricted ranges of depth. Therefore, volume scattering is specified in SST by defining a list of **ReverbLayer** objects, each of which specifies the volume scattering strength (assumed constant) between specified upper and lower depth limits. Currents are specified by giving the average horizontal velocity of the scatterers in each layer.

Currently, SST's reverberation model (Sec. 10) treats each layer as if all of the scatterers were concentrated in a thin sheet in the center of the layer.

7 Sonar and Source Models

The sonar model (the box labeled "Sonar Receiver" in Fig. 1) describes any compact object that can receive sound from the ocean. The source model (the boxes labeled "Sonar Transmitter" and "Source") describes any compact object that emits sound into the ocean. These are represented in SST by objects of classes **Sonar** and **Source**, respectively. For passive sonar scenarios the **Source** is whatever the sonar is listening to. For active sonar scenarios the **Source** is the transmitter. Other **Source** objects may represent countermeasures, the sonar's own vehicle, interfering ships in the area, pile drivers, explosions, or anything else that produces sound from a small region. **Sonar** and **Source** objects are also used internally to model target echoes (Sec. 9).

Classes **Sonar** and **Source** are almost identical. Each object of either of those classes contains the following attributes:

- *trajectory*: an object of a class derived from base class **Traject**. It determines the location and orientation of the sonar or source platform as a function of time.

- *beams*: a list of objects derived from base class **Beam**. They determine the spatial properties of each channel, including its directional sensitivity and the location of its phase center.

- *signal*: an object of a class derived from base class **Signal**. For a **Source**, this is an input representing transmitted sound. For a **Sonar**, it is an output representing received sound. (The association of a **Signal** with a **Sonar** is implicit; it is not formally an attribute of class **Sonar**.)

An active sonar system consists of a **Sonar** and at least one **Source**. It is *monostatic* if those components share a single trajectory, and it is *bistatic* if they have different trajectories.

7.1 Trajectories and Coordinate Transformations

In **Sonar**, **Source**, and **Target** objects, the local geometric properties (the locations and beam patterns of the transducers, sources, or highlights) are expressed in a *platform-centered* coordinate system whose components are in the directions (Forward, Starboard, Below) relative to some arbitrarily defined "center" of a particular platform (its *origin*). Global properties of the ocean are expressed in the *Earth-centered* coordinate system, whose components are in the directions (North, East, Down) relative to an origin at an arbitrary point on the surface of the ocean. The time-dependent mapping between the local and global coordinate systems is defined by the *trajectory* attribute of each vehicle. An SST user specifies the trajectory by assigning to this attribute an object of any subclass of class **Traject** — usually a **Trajectory**, but sometimes a **CombinedTraject**.

Two kinds of vectors need to be transformed between local and coordinate systems: locations and directions (slowness vectors). These transformations take the following linear form:

$$\begin{aligned} \boldsymbol{r}' &= \boldsymbol{M}_X(t)\ (\boldsymbol{r} - \boldsymbol{r}_X(t)) \\ \boldsymbol{S}' &= \boldsymbol{M}_X(t)\,\boldsymbol{S}\,, \end{aligned} \tag{37}$$

where the primed vectors are expressed in platform coordinates and the unprimed vectors are in Earth-centered coordinates. The vector $\boldsymbol{r}_X(t)$ is the location of the origin of platform X, and $\boldsymbol{M}_X(t)$ is the 3-by-3 rotation matrix determined by the orientation of the platform (sonar, source, or target). Each vehicle's trajectory supplies the time-dependent location $\boldsymbol{r}_X(t)$ and rotation matrix $\boldsymbol{M}_X(t)$ used in these coordinate transformations.

A **Trajectory** (the most commonly used subclass of **Traject**) specifies a body's motion using a list of **Snapshot** objects. Each **Snapshot** is a "picture" of the position, velocity, orientation, and rotation rate of a body (four attributes, three numbers each) at a single specified time. For intermediate times the trajectory is computed using self-consistent cubic interpolation, in which all four attributes are continuous in time. For times outside the range of the list, each **Trajectory** is extrapolated using the assumption that the velocity and the rotation rate remain constant in the body's own coordinate system. Thus, a **Trajectory** containing a single **Snapshot** can be used to specify motion in a straight line, a circle, or a helix.

The other **Traject** subclass is **CombinedTraject**, which is specified in terms of two other **Traject** objects: one to specify the motion of a body with respect to an intermediate coordinate system, and another to specify the motion of that intermediate coordinate system with respect to a global system. A **CombinedTraject** can be

used anywhere that a **Traject** is required, but its primary application is internal, in the implementation of SST's target models.

Internally, orientations and rotations are represented using *quaternions* [Dean 1966], also known as *Cayley-Klein parameters* [Goldstein 1950]. This representation is chosen for its combination of compactness with efficient support of coordinate transformations, composition of rotations, interpolation, and extrapolation.

7.2 Beam Patterns

Each **Sonar** and each **Source** contains an attribute called *beams*, which consists of a user-specified list of objects belonging to classes derived from the base class **Beam**, one object per channel in the associated *signal*. Each **Beam** object provides member functions that compute the following quantities:

- $B_c(f, \boldsymbol{S}')$: the directional sensitivity pattern used in Eq. (39) (or the corresponding one for source beams), given the frequency f and the slowness vector $\boldsymbol{S}'$ in platform coordinates. For element-level simulations, this is the sensitivity pattern of one element, as modified by the physical supports and baffles surrounding it and by any preamplifiers or filters between the element and the injection point chosen for the simulation. For beam-level simulations, this is the effective sensitivity pattern of the array plus surrounding hardware, as modified by any signal processing steps from the array through the beamformer.
- $\boldsymbol{r}'_c$: the channel offset. This is the location, in platform coordinates, of the phase center of channel c. For element-level simulations, this is the location of a transducer relative to the array center. For beam-level simulations, it is often zero, or sometimes the center of a sub-array used to form an "offset phase center" beam.
- $\tau_p(\boldsymbol{S}', \boldsymbol{r}'_c)$: the offset delay used in Eq. (31). The expression used is essentially Eq. (30), using the channel offset $\boldsymbol{r}'_c$ in place of $\boldsymbol{\rho}_R$ and with inputs in platform coordinates.

Subscript c stands for the receiver channel r or the source channel s.

The various subclasses of class **Beam** differ from one another in the algorithm used to compute $B_c(f, \boldsymbol{S}')$. They are listed in Table 6.

The first section in the table contains simple, self-contained beam patterns, most of which are based on equations in Chapter 3 of [Urick 1983]. For example, **StickBeam**, **PistonBeam**, and **LineBeam** come from the first three rows of Urick's Table

Table 6: Beam Pattern Models

Class	*Summary*
Beam	Abstract base class
OmniBeam	Omnidirectional (1.0 everywhere)
BinomialBeam	Binomial weighted, steered line array
ConeBeam	1.0 inside a cone, 0 outside
DCLineBeam	Dolph-Chebyshev weighted line array [Albers 1965]
ElementSumBeam	Sum of weighted elements beam pattern
LineBeam	Uniformly weighted, steered line array
PistonBeam	Circular piston transducer
RecPistonBeam	Rectangular piston transducer
StickBeam	Continuous, uniform line transducer
EBFTableBeam	Interpolated from table vs. elevation, bearing, frequency
EFIntensityBeam	Intensity vs. sin(elevation) and frequency
SIOBeam	Table vs. elev, bear, freq in binary SIO file
DecibelBeam	Beam pattern transformed from decibels to pressure ratio
ProductBeam	Product of input beam patterns
RotatedBeam	Beam pattern rotated with respect to the platform coordinates
SumBeam	Sum of element beam patterns
WeightedBeam	Beam pattern multiplied by a weight and phase-shifted for delay

3.2 (second column, omitting the squaring operation). Each beam pattern object is constructed using input parameters that differ from class to class. For example, **PistonBeam** requires a piston diameter and axis direction, whereas **LineBeam** requires the number and spacing of the elements, axis direction, and steering delay.

The classes in the second section in the table accept tables of numbers; $B_c(f, \boldsymbol{S}')$ is computed by interpolation. **EFIntensityBeam** and **SIOBeam** optionally accept another beam pattern as input, in which case the table is computed by sampling the input pattern; the result may be used in this or subsequent runs to speed up the simulation.

The classes in the third section of the table are transformations that accept one or more input beam patterns and transform them in some way.

All beam patterns accept, in addition to their class-specific input parameters, an offset vector $\boldsymbol{r}'_c$ and an additional delay to be added to the offset delay $\tau_p(\boldsymbol{S}', \boldsymbol{r}'_c)$.

7.3 Sonar Transformation

The sonar model has a more abstract interpretation: it represents a transformation whose output is the signal $y_r(t)$ in all channels r of receiver R, and whose input is the sound field in the water, $p_R(t, \boldsymbol{r})$, for $\boldsymbol{r}$ in the neighborhood of the sonar origin $\boldsymbol{r}_R$. Such a sound field is produced as the output of the eigenray transformation (Eq. (31)). The sonar transformation takes the following form (in the time domain and platform coordinates):

$$y_r(t) = \int \int b_r(\tau, \boldsymbol{\rho}'_r)\, p'_R(t - \tau, \boldsymbol{r}'_r + \boldsymbol{\rho}'_r)\, \mathrm{d}\tau\, \mathrm{d}\boldsymbol{\rho}'_r\,, \tag{38}$$

where $\boldsymbol{r}'_r$ is the channel offset vector provided by the beam pattern model for receiver channel r (Sec. 7.2). The time-space domain kernel $b_r(\tau, \boldsymbol{\rho}'_r)$ for receiver channel r is related to the receiver's beam pattern $B_r(f, \boldsymbol{S}')$ by a four-dimensional spatiotemporal Fourier transform:

$$b_r(\tau, \boldsymbol{\rho}'_r) = \int \int B_r(f, \boldsymbol{\nu}'/f)\, \mathrm{e}^{2\pi \mathrm{i}(f\tau + \boldsymbol{\nu}' \cdot \boldsymbol{\rho}'_r)}\, \mathrm{d}f\, \mathrm{d}\boldsymbol{\nu}'\,, \tag{39}$$

where $\boldsymbol{\nu}'$ is the wave number vector in vehicle coordinates. The spatial kernel $b_r(\tau, \boldsymbol{\rho}'_r)$ is nonzero over the region $\boldsymbol{\rho}'_r$ where the receiver is sensitive. For an ideal piston beam the sensitive region is the disk at the face of the transducer, although in practice effects like baffling and shadowing tend to make it more complicated. SST starts from the frequency-direction form, $B_r(f, \boldsymbol{S}')$, where $\boldsymbol{S}' = \boldsymbol{\nu}'/f$ (beam patterns versus frequency and look direction).

The primed coordinates in Eqs. (38) and (39) are expressed in a platform-centered coordinate system whose origin is at $\boldsymbol{r}_R(t)$. Equation (37) defines their relationship to the unprimed, Earth-centered coordinates used in the eigenray transformation (Eq. (31)).

7.4 Source Transformation

The source model has a similarly abstract interpretation: it represents a transformation whose input is the signal $x_s(t)$ emitted through each channel s of the source S, and whose output is the sound source field $q_S(t, \boldsymbol{r}_S + \boldsymbol{\rho}_S)$ in the neighborhood of $\boldsymbol{r}_S$. This sound source field is the input of the eigenray transformation (Eq. (31)). The source transformation takes the following form (in the time domain):

$$q_S(t, \boldsymbol{r}_S + \boldsymbol{\rho}_S) = \sum_s \int b'_s(\tau, \boldsymbol{M}_S(t)\, \boldsymbol{\rho}_S - \boldsymbol{r}'_s)\, x_s(t - \tau)\, \mathrm{d}\tau\,, \tag{40}$$

where $\boldsymbol{r}'_s$ is the channel offset vector provided by the beam pattern model for source channel s (Sec. 7.2). The time-space domain kernel $b'_s(\tau, \boldsymbol{\rho}'_s)$ for source channel s is related to the source's beam pattern $B_s(f, \boldsymbol{S}')$ as per the receiver (Eq. (39)) except for the sign in the relationship $\boldsymbol{S}' = -\boldsymbol{\nu}'/f$. The relations between platform-centered (primed) and Earth-centered (unprimed) coordinates are given by Eq. (37).

8 Direct Sound Propagation Models

The "direct" sound propagation model is a transformation whose input is the sound $x_s(t)$ emitted by all channels s of a given source, and whose output is that portion of the received sound $y_r(t)$ that does not scatter from objects or irregularities on its way to receiver channel r. Conceptually, it consists of the successive application of the source model, the eigenray model, and the receiver model, whose time-domain expressions are given by Eqs. (40), (31), and (38), respectively. If we combine those equations in series, the spatial integrations reduce to Dirac delta functions and drop out. The remaining operations can be arranged into successive signal transformations, as follows:

$$x_{sp}(t) = x_s\left(t - T_p - \tau_p(\boldsymbol{M}_S \boldsymbol{S}_{Sp}, \boldsymbol{r}'_s)\right) \tag{41}$$

$$x_p(t) = \sum_s \int \left[\int \mathrm{e}^{2\pi \mathrm{i} f_S \tau'_S} B_s(f_S, \boldsymbol{M}_S \boldsymbol{S}_{Sp}) \mathrm{d}f_S \right] x_{sp}(t - \tau'_S)\, \mathrm{d}\tau'_S \tag{42}$$

$$y_p(t) = \int \left[\int \mathrm{e}^{2\pi \mathrm{i} f \tau} L_p(f) \mathrm{d}f \right] x_p(t - \tau)\, \mathrm{d}\tau \tag{43}$$

$$x_{rp}(t) = \int \left[\int \mathrm{e}^{2\pi \mathrm{i} f_R \tau'_R} B_r(f_R, \boldsymbol{M}_R \boldsymbol{S}_{Rp}) \mathrm{d}f_R \right] y_p(t - \tau'_R)\, \mathrm{d}\tau'_R \tag{44}$$

$$y_{rp}(t) = x_{rp}\left(t - \tau_p(\boldsymbol{M}_R \boldsymbol{S}_{Rp}, \boldsymbol{r}'_r)\right) \tag{45}$$

$$y_r(t) = \sum_p y_{rp}(t). \tag{46}$$

Thus, the transformation of a source channel signal $x_s(t)$ to a receiver channel signal $y_r(t)$ involves three filters [source beam pattern $B_s(f, \boldsymbol{S}'_{Sp})$, eigenray loss $L_p(f)$, and receiver beam pattern $B_r(f, \boldsymbol{S}'_{Rp})$] plus three delays [source channel offset $\tau_p(\boldsymbol{S}'_{Sp}, \boldsymbol{r}'_s)$, eigenray T_p, and receiver channel offset $\tau_p(\boldsymbol{S}'_{Rp}, \boldsymbol{r}'_r)$], plus sums over eigenrays and source channels. The offset delays are given by Eq. (30). The three inverse Fourier transforms (in square brackets) compute the time-domain impulse responses for the filters, and the three time integrations convolve those impulse responses with the signal.

Note that all of the filters and delays depend parametrically on time t because the trajectory attributes $\boldsymbol{r}_R$, $\boldsymbol{r}_S$, $\boldsymbol{M}_R$, and $\boldsymbol{M}_S$ depend on time (the sonar and source can move), and the eigenray attributes $\boldsymbol{S}_{Sp}$, T_p, $L_p(f)$, and $\boldsymbol{S}_{Rp}$ depend on the eigenray's end points $\boldsymbol{r}_R$ and $\boldsymbol{r}_S$.

8.1 DirectSignal

Equations (41) through (45) represent "ideal" filters and delays. Their implementation by SST class **DirectSignal** looks very much like those equations, except that the convolutions and interpolations are forced to have finite length. For each path p, **DirectSignal** sets up a chain of five "data flow" objects (Sec. 4.3) in series: two **VarDelay** objects (Sec. 4.5.1) implementing Eqs. (41) and (45), sandwiching three **VarFirFilter** objects (Sec. 4.5.2) implementing Eqs. (42), (43), and (44). The chains p feed into a **SumSignal** object implementing Eq. (46). The **VarDelay** and **VarFirFilter** objects are fed by lower-volume internal data flow objects carrying the time-varying delays and filter responses from the eigenray model and the source and receiver models.

This entire network of interconnecting objects is set up by the *openRead* operation at the start of the **CopySignal** operation. In the **CopySignal** main loop,each *readBlock* operation activates all of the objects in that network as required to compute that block. At the end of the **CopySignal** operation, the *close* operation of **DirectSignal** shuts down and destroys the network of data flow objects.

8.2 DirectSpectrum

As the signal bandwidth increases, the lengths of the required FIR filters may increase to the point where it becomes advantageous to do all three filters (Eqs. (42) through (44)) in the frequency domain. To do this, SST offers class **DirectSpectrum** as an alternative to **DirectSignal**. **DirectSpectrum** produces its results in the windowed frequency domain form as defined by Eq. (7), as follows:

$$\begin{aligned} Y_r(f,t) \;=\; & \sum_p B_r(f, \boldsymbol{M}_R \boldsymbol{S}_{Rp})\, L_p(f) \sum_s B_s(f, \boldsymbol{M}_S \boldsymbol{S}_{Sp}) \\ & \times X_s(f, t - T_p - \tau_p(\boldsymbol{M}_S \boldsymbol{S}_{Sp}, \boldsymbol{r}'_s))\,. \end{aligned} \tag{47}$$

For each eigenray p the initial delay, which includes the source channel offset delays $\tau_p(\boldsymbol{M}_S \boldsymbol{S}_{Sp}, \boldsymbol{r}'_s)$ and the eigenray delay T_p, is done in the time domain using class **VarDelay**, just as it is in **DirectSignal**. Following that delay **DirectSpectrum** transforms the signal into the windowed frequency domain form (class **SpectrumFromSignal**, Eq. (7)). The three successive filters (source beams, eigenray, and receiver

beams) are done in the frequency domain, and the result remains in the frequency domain. For cases where the filters are long, this saves the time needed to transform back and forth between time and frequency domains between filters. Unfortunately, SST does not (yet) have a frequency-domain delay operation, so **DirectSpectrum** cannot be used if the receiver channels have offsets. **DirectSpectrum** was used heavily in a passive-sonar application several years ago, and (to my knowledge) has not been used since.

9 Target Echo Model

SST class **TargetEcho** is a data-flow class (Sec. 4.3) whose attributes include a **Source**, a **Sonar**, and a **Target**. The output represents sound that has been transmitted by the **Source**; received, altered, and re-transmitted by the **Target**; and finally received by the **Sonar**.

From **TargetEcho**'s point of view, a target consists of a group of receivers (**Sonar** objects) and a group of transmitters (**Source** objects) back to back, with a target-specific transformation that determines what happens to the received sound before it is re-transmitted.

Class **TargetEcho** is implemented using one or more **DirectSignal** objects to carry sound from the source to the target, and one or more **DirectSignal** objects to carry sound from the target to the receiver. The transformation from the sound received by the **Target** to the sound re-transmitted by the **Target** is determined by the **Target** subclass. The **TargetEcho** object sets up the network of **DirectSignal** objects at the start of a **CopySignal** operation, and tears it down at the end of that operation.

This algorithm is inherently *bistatic* — nowhere is it assumed that the source and the receiver are at the same location. The common *monostatic* case is modeled by assigning the same trajectory to both the source and the receiver. The code that SST executes is exactly the same for either monostatic or bistatic systems. In fact, the source need not be a conventional active transmitter, and the signal need not be a pulse. This flexibility enables SST to model tactically interesting effects such as scattering of sound emitted by a broadband jammer countermeasure from the submarine that launched it, as heard by another submarine. That situation is not what is conventionally thought of as a "target echo", but SST's **TargetEcho** class will handle it.

9.1 Target Models

All of SST's target models are derived from the base class **Target**. Every **Target** contains a trajectory (Sec. 7.1). Each subclass of **Target** adds other attributes, which determine the relationship between the sound received by the target and the sound it re-transmits. All of the existing subclasses of **Target** express that relationship in terms of *highlights*, which are point scatterers that move as a group along the target's trajectory. However, the **Target** interface used by **TargetEcho** (summarized above) is general enough to support not only highlight-based target models but also artificial targets and active countermeasures.

The three existing target models differ in how the placement and properties of their highlights are determined.

9.1.1 PointTarget

Class **PointTarget** is SST's simplest target model. The user specifies a list of point scatterers, each of which is characterized by its scattering strength and its position and velocity relative to the target's local coordinate system. The signal transmitted from each highlight's location is simply a scaled copy of the signal received at that highlight.

The user can choose between *common center* processing and *multiple center* processing. In the *common center* case the **PointTarget** creates only one receiver and one transmitter located at the target's origin, each of which has multiple channels, one per highlight. This causes **TargetEcho** to create only two **DirectSignal** objects, with multiple channels at the target end. In the *multiple center* case the **PointTarget** creates a separate, single-channel receiver and transmitter for each highlight. This causes **TargetEcho** to create two separate **DirectSignal** objects for each highlight.

The *common center* option is always faster, but it introduces an additional approximation: the offsets from the target center to the individual highlight locations affect only the arrival times of the echoes, and not the directions from which they arrive. Therefore, the *multiple center* option should be chosen whenever the target is close enough for the sonar system to detect and use the target's cross-range extent (e.g., to compute "line-like" classification clues).

PointTarget also allows the user to specify a *randomPosition* value, which gives the root mean square value of a Gaussian random component to be added to the specified highlight locations. This can be used to break up unrealistic grating effects from regularly spaced highlights.

9.1.2 HighlightTarget

Class **HighlightTarget** improves on **PointTarget** in three ways:

- Each highlight may have a complex, frequency-dependent response.
- Each highlight may have a delay between reception and transmission.
- Each highlight is assigned a *group number*. All highlights with the same group number are treated as channels in a single receiver-transmitter pair. This provides the flexibility to model cross-range extent (e.g., by placing stern highlights in one group and bow highlights in another) without paying the performance price of treating each highlight as a separate target. Assigning each highlight a different group number is equivalent to setting *commonCenter* to *false* in **PointTarget**. Assigning the same group number to all highlights is equivalent to setting *commonCenter* to *true* in **PointTarget**.

9.1.3 ExternalTarget

Class **ExternalTarget** is functionally identical to **HighlightTarget** except for the source of its highlights. When you create an **ExternalTarget** object, it starts up an external program provided by the user. At the start of each ping (when the **TargetEcho** object is opened for reading), the **ExternalTarget** object sends to the external program the distance and direction to the active sonar's transmitter and receiver and the range of signal frequencies. The external program sends back a list of highlights. Once that exchange is complete, **ExternalTarget** behaves exactly like a **HighlightTarget**.

This information exchange occurs via a pair of pipes connected to the external program's standard input and standard output streams. The protocol involves text commands, and the data are in text form. Hence the external program can be tested in isolation, without SST. The SST distribution includes a Fortran skeleton to serve as a starting point, or the protocol can be implemented in the user's favorite language.

Existing target models in use by the Navy take the form of subroutines that return lists of highlights. Several of these models have been wrapped for use with SST using class **ExternalTarget**. Among other advantages, this allows simulators to use classified target models while keeping the SST code unclassified.

10 Reverberation

Reverberation can be thought of as the sum of echoes from a very large number of discrete scatterers, each of which sends back a delayed, filtered, Doppler-shifted copy of the source signal. Other descriptions are also valid, and mostly equivalent. Reverberation scatterers are on the surface (waves and bubbles), the bottom (roughness and embedded inhomogeneities), and the ocean volume (mostly marine life). The physical assumptions are the same as those used for target echoes: linearity, eigenray propagation, single scattering, and restriction to the far field of the transmitter and receiver.

A few more assumptions are added for reverberation: the scatterers that contribute to reverberation are randomly distributed, and their density is so high that the sonar system cannot resolve them as individual scatterers. Further, the number of scatterers in each resolution cell of the sonar is assumed high enough to reach the Gaussian limit. The consequence is that *reverberation has Gaussian statistics.*

Certain tactically important phenomena that are conventionally regarded as reverberation, most notably rock outcrops, often do not satisfy that assumption. For now this problem is defined away by saying, "Reverberation is Gaussian. Echoes that are not Gaussian are target echoes." If false targets are tactically important, the only current recourse is to model them explicitly as targets. We are working on other options.

One way to generate simulated reverberation is the "point scatterer" approach: Generate a very large number of discrete random scatterers with random locations, velocities, and sizes, generate target echoes for each one, and add them up. If enough scatterers are used in each sonar resolution cell, the resulting signal will approach a Gaussian distribution by virtue of the Central Limit Theorem. Early versions of SST [Goddard 1989] and REVGEN [Princehouse 1975, Princehouse 1978, Goddard 1986] used variants of that approach. Unfortunately, as the system time-bandwidth product increases, the point scatterer approach becomes impractical, both because the required density of scatterers increases and because frequency dependence in the echo model becomes important.

SST's current reverberation model is based on the *scattering function approach*, which is less direct and obvious than the point scatterer approach, but much more efficient. The task of generating simulated reverberation is broken into two major steps:

- Compute the *scattering function*, a generalized intensity impulse response function introduced in Sec. 4.2.2.

- Combine the scattering function with the source signal and a stream of random numbers to generate a realization of the reverberation signal.

This approach is not new. The basic idea has been described by Luby and Lytle [Luby Lytle 1987], Hodgkiss [Hodgkiss 1984], and Chamberlain and Galli [Chamberlain Galli 1983], all of which built on the classic work of Faure [Faure 1964], Ol'shevskii [Ol-shevskii 1967], and Middleton [Middleton 1967]. SST's implementation is applicable to far more real systems than the versions described in those references because we have discarded most of the simplifying assumptions: short narrowband pulses, isovelocity single-path propagation, monostatic geometry, and others.

10.1 Generating Reverberation

We begin with the second of the two steps: given a scattering function and a source signal, generate a stochastic realization of the received reverberation.

SST computes the power spectral density for reverberation by performing a two-dimensional convolution (versus Doppler Γ and two-way travel time T) of the scattering function $Z_{rr's}(f, \Gamma, T)$ with the power spectral density of the transmit signal:

$$P_{rr's}(f, t_u) = \int \int \Gamma^{-2} \, Z_{rr's}(\Gamma, f, t_u - t_{u'}) \, P_s(f/\Gamma, t_{u'}) \, \mathrm{d}\Gamma \, \mathrm{d}t_{u'} \, . \tag{48}$$

The source PSD $P_s(f, t_u)$ comes from applying Eqs. (7) and (12) to the source signal.

Starting with this estimate for the PSD, SST generates Gaussian random realizations of the reverberation using the same Mitchell-McPherson algorithm used for Gaussian noise, described in Sec. 4.6.1. The validity of this step depends on the same assumptions required for noise: that the statistics are Gaussian and that an update interval Δ exists such that the time variation of $P_{rr's}(f, t_u)$ is slow on a scale of Δ and, simultaneously, the frequency variation is slow on a scale of $1/\Delta$. The resulting signal has a very narrow frequency coherence width (order $1/\Delta$) and a coherence time that is usually of the same order as the inverse of the signal bandwidth but no longer than Δ. This is realistic if the scattering function is sufficiently smooth.

SST Classes: Class **ReverbSignal** generates a realization of Gaussian reverberation starting from the scattering function and the transmit signal. It is implemented using an object of class **ReverbSpectrum**, which implements Eq. (48), together with **SpectrumFromSignal** [to compute $P_s(f, t_u)$, Sec. 4.2] and the Mitchell-McPherson noise generation classes **FactorSpectrum**, **GaussianSpectrum**, and **SignalFromSpectrum** (Sec. 4.6.1).

10.2 Computing the Scattering Function

A detailed derivation of the scattering function, and a discussion of the approximations and assumptions on which it is based, are beyond the scope of this paper. Some of those issues are discussed in a technical report [Goddard 1993] written early in SST's development. We are preparing a paper to update that material and add new material on control of spatial and temporal coherence. Here, we will skip the derivation and go directly to the answer.

The scattering function $Z_{rr's}(\Gamma, f, T)$ is computed by evaluating the following integral:

$$\begin{aligned} Z_{rr's}(\Gamma_k, f_m, T_\nu) &= \frac{1}{H\left(\delta\Gamma_k\right)} \sum_{plq} \int_{A_{k\nu,plqs}} U_{plqs}(f_m, \boldsymbol{r}) \\ &\quad \times B_r(f_m, \boldsymbol{M}_R \boldsymbol{S}_{Rp}(\boldsymbol{r}))\, B^*_{r'}(f_m, \boldsymbol{M}_R \boldsymbol{S}_{Rp}(\boldsymbol{r}))\, \mathrm{e}^{2\pi \mathrm{i} f_m \tau_{rr'p}(\boldsymbol{r})}\, \mathrm{d}\boldsymbol{r}\,, \end{aligned} \tag{49}$$

where

$$\begin{aligned} U_{plqs}(f_m, \boldsymbol{r}) &= \left|B_s(f_m, \boldsymbol{M}_S \boldsymbol{S}_{Sq}(\boldsymbol{r}))\right|^2 \left|L_p(f_m, \boldsymbol{r}_R, \boldsymbol{r})\right|^2 \left|L_q(f_m, \boldsymbol{r}, \boldsymbol{r}_S)\right|^2 \\ &\quad \times S_l(f_m, \boldsymbol{S}_{Tp}(\boldsymbol{r}), \boldsymbol{S}_{Tq}(\boldsymbol{r})) \end{aligned} \tag{50}$$

and $\tau_{rr'p}(\boldsymbol{r})$ is the difference between the first-order offset delays for receiver channels r and r':

$$\tau_{rr'p}(\boldsymbol{r}) = (\boldsymbol{M}_R \boldsymbol{S}_{Rp}(\boldsymbol{r})) \cdot (\boldsymbol{r}'_r - \boldsymbol{r}'_{r'}). \tag{51}$$

The other factors in the integrand are familiar: the beam patterns $B_x(f, \boldsymbol{S}')$, the propagation losses $L_p(f, \boldsymbol{r}_R, \boldsymbol{r}_S)$, and the bistatic scattering strength $S_l(f, \boldsymbol{S}_p, \boldsymbol{S}_q)$. The sum in Eq. (49) is over receive-path eigenrays p connecting the scattering field at $\boldsymbol{r}$ to the receiver, transmit-path eigenrays q connecting the source to $\boldsymbol{r}$, and a scattering layer indexed by l. These "layers" include the surface, the bottom, and any number of volume scattering layers. SST treats volume scattering layers as if all of the scatterers were concentrated on a thin sheet at the center of the layer; this reduces the spatial integration from three dimensions to two.

The scattering function has three conceptually continuous independent variables, all of which are divided into bins for sampling: the Doppler shift Γ, the frequency f, and the two-way travel time T. It also has three discrete indices: the source (or source channel) s and two receiver channels r and r'. The appearance of two receiver channels underscores the nature of the scattering function as a second-order statistic; the off-diagonal elements ($r \neq r'$) give rise to correlations between receiver channels.

The subtle part is defining the domain of integration. For each layer and pair of eigenrays, the domain of integration $A_{k\nu,plqs}$ in Eq. (49) is defined implicitly as the

locus of locations $\boldsymbol{r}$ on layer l for which the total round-trip propagation time falls within time bin ν (centered on T_ν with width H) and for which the Doppler shift due to platform motion falls within Doppler bin k (centered on Γ_k with width $\delta\Gamma_k$) for a given receive-leg path p, scattering layer l, and transmit-leg path q.

For a straight-line propagation model with a flat or uniformly sloping bottom, the locus of constant two-way travel time is an ellipse. For a general eigenray model, there is no closed-form mapping from time and Doppler to location, so SST uses standard numeric root-finding techniques to define an approximately elliptical integration path and to partition it into segments that fall into each Doppler bin. The details are beyond the scope of this paper.

SST Classes: SST provides two classes, **BBBScatFun** and **BBBDirectionalScat**, which differ primarily in the strategy used to partition the scattering layers into time-Doppler bins. **BBBScatFun** does the integration of Eq. (49) separately for each eigenray pair and layer *plq*, and sums them. The partitioning and integration are done with considerable care for continuity and accuracy. Evaluation of the receive beam patterns is in the inner loop; the number of beam pattern evaluations scales with the product of the square of the number of beams, the square of the number of eigenrays, the number of scattering layers, and the number of source channels.

The second choice, **BBBDirectionalScat**, gains speed (sometimes) by factoring out the receive beam patterns and taking less care with the numerics. The first stage is computing the *directional scattering function*:

$$Z_s(\Gamma_k, f_m, \theta_i, \phi_j, T_\nu) = \frac{1}{H\,(\delta\Gamma_k)} \sum_{plq} \int_{A_{kij\nu,plqs}} U_{plqs}(f_m, \boldsymbol{r})\,\mathrm{d}\boldsymbol{r}\,, \tag{52}$$

where now the integrand (Eq. (50)) excludes the receive beam patterns, and the integration area $A_{kij\nu,plqs}$ is a patch on the scattering layer such that the eigenray direction at the receiver falls in a small cell centered on elevation angle θ_i and bearing angle ϕ_j in receiver-centered coordinates. For a given time bin T_ν, this large data structure is accumulated by evaluating the integrand for each ray pair and layer *plq* at a number of sample points around the locus of constant travel time T_ν, and adding a properly scaled increment to the closest cell in Γ, θ, and ϕ for all frequencies f_m.

The directional scattering function is almost always very sparse. For a given source s and time T_ν, four dimensions remain. The contribution for a given eigenray pair and layer *plq* falls on a thin, closed band on the unit sphere, and those bands move toward the equator (low elevation) as travel time increases. For a given direction cell ij, typically only one Doppler cell k is nonzero. **BBBDirectionalScat** uses a compromise memory management scheme that avoids wasting memory without a large performance penalty.

SST includes hooks and Matlab scripts to create a Matlab movie that shows how the directional scattering function evolves with time.

The second stage of **BBBDirectionalScat** inserts the receiver beam patterns and the phase shift due to receiver channel offsets:

$$\begin{aligned} Z_{rr's}(\Gamma_k, f_m, T_\nu) &= \sum_{ij} (\delta\theta_i)\,(\delta\phi_j)\, Z_s(\Gamma_k, f_m, \theta_i, \phi_j, T_\nu) \\ &\quad \times B_r(f_m, \boldsymbol{S}_{ij})\, B^*_{r'}(f_m, \boldsymbol{S}_{ij})\, \mathrm{e}^{2\pi \mathrm{i} f_m \boldsymbol{S}_{ij}\cdot(\boldsymbol{r}'_r - \boldsymbol{r}'_{r'})}\,, \end{aligned} \tag{53}$$

where $\boldsymbol{S}_{ij}$ is the slowness vector corresponding to elevation θ_i and bearing ϕ_j in receiver-centered coordinates. The cell sizes in θ and ϕ are chosen by the user to be small compared to the resolution of the sonar. The beam patterns are evaluated only once for the required angles and frequencies, and used for all time steps.

The advantage of **BBBDirectionalScat** is that it is often faster than **BBBScatFun**, especially when the sonar has many channels and there are many eigenrays (e.g., in shallow water). The main disadvantage is that the resulting scattering function tends to be somewhat jagged; neighboring direction cells may have very different values depending on where the integration sample points happen to fall. Users are advised to start with **BBBScatFun** for its more careful numeric techniques. Those who switch to **BBBDirectionalScat** for speed should compare the results of the two methods and adjust the sample density until they give close to the same answers. Of course, only the user can judge how close is close enough.

11 Summary and Plans

The Sonar Simulation Toolset is a mature, well-supported software product that has contributed to many Navy projects since its first release in 1989. It produces sound, suitable for listening or for feeding into a sonar front end, using commonly available computers. SST is portable, general, broadband, bistatic, multi-channel, embeddable, streamable, object oriented, unclassified, and offers flexible fidelity. It is available to any DoD agency or contractor.

SST is actively supported and continuously improving. Development is always driven by users' requirements and sponsors' priorities. We conclude with some directions that SST may move from here.

Coherence Control: In SST 4.1, the version described in this report, forward propagation (Sec. 8) is entirely distinct from reverberation (Sec. 10). Forward specular reflections can involve loss of energy but no loss of coherence, whereas the

reverberation model produces almost complete loss of coherence. The truth is somewhere between those extremes. The solution is to integrate the forward reflection and forward scattering models into a single, consistent model that generates FAT (Frequency, Angle, Time) spreading for all forward-propagated signals. Under current funding from ONR 321US, SST soon will include an initial model for FAT spreading, based in large part on work by Dahl [Dahl 1996, Dahl 1999, Dahl 2001, Dahl 2002]. Applications will include design and evaluation of longer, higher-bandwidth active transmit signals, LPI pulses, and acoustic communications.

Performance Tuning and Parallel Processing: SST is not intended as real-time software. Nevertheless, speed is important to users. Current funding from ONR 333 includes performance tuning, including work toward parallel algorithms based on the MPI library [Gropp Lusk Skjellum 1999]. This will provide dramatic performance increases for multi-processor computers and Beowulf-style [Beowulf] clusters of commodity computers.

Target-like Clutter: Gaussian reverberation does not tell the whole story; a statistical model of target-like bottom clutter will be required for realistic performance prediction in many important shallow-water environments. Such models are under development [Abraham Lyons 2002]. Current work funded by ONR 333 will bring statistical clutter models into SST.

Interoperability: SST lies in the middle of the modeling and simulation hierarchy because it is both a consumer of other people's models and a producer of signals for input to higher-level simulations like TRM. Hence, portability and interoperability are especially important concerns. The TEAMS [TEAMS] (and other) standardization efforts will guide us in making SST more open, both from below and from above.

Wakes: Ship wakes are a huge, complex problem, especially from the point of view of torpedo defense. Current funding from ONR 321US supports simple modeling of wakes, but that is only a start. Fundamental, innovative changes to current modeling approaches will be necessary to model wakes more accurately and comprehensively.

Updating Component Models: Improving the fidelity of SST's underlying models is a continuing task, driven both by improvements in our scientific understanding and by the requirements of new SST applications. Examples of improvements that should be included in SST are the small-slope approximation for rough-surface scattering [Thorsos Broschat 1995] and elastic bottom scattering [Jackson Ivakin 1998]. In addition, SST's support for very long passive

scenarios is incomplete, and SST's practice of treating volume scattering layers as thin sheets of scatterers produces artifacts early in the return in some situations. These and similar shortcomings must be corrected.

Lower Frequencies: Users in the air ASW community and others want SST-like signal level simulation tools for lower frequencies. A bottom model with penetration to sub-bottom layers would be a good start. A more radical change would be to incorporate a wave-based propagation model.

Verification, Validation, and Accreditation: Making sure that SST simulations reflect reality with useful accuracy is a continuing effort. We are always looking for challenging data sets that are sensitive tests of SST's unique capabilities.

Support: We support SST users by answering their questions, helping them set up simulations, tracking down mysteries, fixing bugs, and distributing releases. In return, our users keep us in touch with what is important or not, what works well or doesn't, what seems easy or hard to do, and what needs fixing or enhancement. User support is one of the most important components in each SST contract.

More Applications: We feel that only a small fraction of the Navy users who could benefit from SST are using it. Our primary reward for the work that we do is to see it make a positive difference to the national defense. Torpedoes, torpedo defense, acoustic communications, shipboard and submarine sonars, bottom-mounted arrays, sonobuoys, and many other classes of systems are candidates for SST simulations. Early idea testing, advanced development, performance prediction, interpretation of experiments, operator training, and many other objectives can be served using realistic simulated sonar signals. We look forward to continuing and expanded service to the U.S. Navy.

REFERENCES

[Abraham Lyons 2002] D. A. Abraham and A. P. Lyons, "Novel Physical Interpretations of K-Distributed Reverberation," IEEE J. Ocean. Eng. **27**, 800-813 (2002). 11

[Albers 1965] V. C. Albers, *Underwater Acoustics Handbook – II* (Pennsylvania State University Press, University Park, PA, 1965), 188-189. 6

[APL Models 1994] "APL-UW High Frequency Ocean Environmental Acoustic Models Handbook," APL-UW TR 9407, Applied Physics Laboratory, University of Washington, Seattle, WA, October 1994. 1.3, 5, 6.4.3

[APL SBU] "Applied Physics Laboratory Sensitive But Unclassified Information", https://www.sbu.apl.washington.edu 1.1

[Beowulf] *Beowulf: Introduction, History, Overview*, NASA, http://beowulf.gsfc.nasa.gov/overview.html. 11

[Chamberlain Galli 1983] S. G. Chamberlain and J. C. Galli, "A Model for Numerical Simulation of Non-Stationary Sonar Reverberation Using Linear Spectral Prediction," IEEE J. Ocean. Eng. **OE-8**, 21-36 (1983). 10

[Correia 1988] E. Correia, "Weapons Assessment Facility (WAF)", Technology Digest, Naval Undersea Warfare Center Division Newport, September 1988, pp. 85-86. 1.2

[Cygwin] *Cygwin*^{TM}, Red Hat, Inc., http://www.redhat.com/software/cygwin/. 1.2, 2.6

[Dahl 1996] P. H. Dahl, "On the spatial coherence and angular spreading of sound forward scattered from the sea surface: Measurements and interpretive model," J. Acoust. Soc. Am. **100**, 748-758 (1996). 11

[Dahl 1999] P. H. Dahl, "On bistatic sea surface scattering: Field measurements and modeling," J. Acoust. Soc. Am. **105**, 2155-2169 (1999). 11

[Dahl 2001] P. H. Dahl, "High-Frequency Forward Scattering from the Sea Surface: The Characteristic Scales of Time and Angle Spreading," IEEE J. Ocean. Eng. **26**, 141-151 (2001). 4.2.2, 11

[Dahl 2002] P. H. Dahl, "Spatial Coherence of Signals Forward Scattered from the Sea Surface in the East China Sea," in *Impact of Littoral Environmental Variability on Acoustic Predictions and Sonar Performance*,edited by N. G.

Pace and F. B. Jensen (Kluwer Academic Publishers, Netherlands, 2002), pp. 55-62. 11

[Dean 1966] R. A. Dean, *Elements of Abstract Algebra* (John Wiley and Sons, New York, 1966). 7.1

[Donelan Hamilton Hui 1985] M. A. Donelan, J. Hamilton and W. H. Hui, "Directional Spectra of Wind-Generated Waves," Phil. Trans. R. Soc. Lond. **A 315**, 509-562 (1985). 5

[Doxygen] D. van Heesch, *Doxygen*, http://www.doxygen.org. 1.1

[Eggen Goddard 2002] C. Eggen and R. Goddard, "Bottom Mounted Active Sonar for Detection, Localization, and Tracking," in *Proceedings Oceans '02 MTS/IEEE* (IEEE Publication 0-7803-7534-3, 2002), Vol. 3, pp. 1291-1298, http://ieeexplore.ieee.org/Xplore/DynWel.jsp. 1.3

[Faure 1964] P. Faure, "Theoretical Model of Reverberation Noise," J. Acoust. Soc. Amer. **36**, 259-268 (1964). 10

[Gilbert 1993] K. E. Gilbert, "A stochastic model for scattering from the near-surface oceanic bubble layer," J. Acoust. Soc. Am. **94**, 3325-3334 (1993). 1.3, 5

[Gnuplot] T. Williams and C. Kelley, *gnuplot*, http://www.gnuplot.info/. 2.2

[Goddard 1986] R. P. Goddard, "REVGEN-4 High-Fidelity Simulation of Sonar Pulses," APL-UW 8505, Applied Physics Laboratory, University of Washington, Seattle, WA, June 1986. 1.3, 10

[Goddard 1989] R. P. Goddard, "The Sonar Simulation Toolset," in *Proceedings Oceans '89, The Global Ocean* (IEEE Publication Number 89CH2780-5, 1989), Vol. 4, pp. 1217-1222. 10

[Goddard 1993] R. P. Goddard, "Simulating Ocean Reverberation: A Review of Methods and Issues," APL-UW 9313, Applied Physics Laboratory, University of Washington, Seattle, WA, October 1993. 10.2

[Goddard 2000] R. P. Goddard, "SST Bistatic Reverberation Modeling: North Sea Experiment, April-May 1998," APL-UW TM 5-00, Applied Physics Laboratory, University of Washington, Seattle, WA, May 2000. 1.3

[Goldstein 1950] H. Goldstein, *Classical Mechanics* (Addison-Wesley, Reading, MA, 1950). 7.1

[Golub Van Loan 1996] G. H. Golub and C. F. Van Loan, *Matrix Computations* (Johns Hopkins University Press, Baltimore, MD, 1996), 3rd ed. 4.6.1

[Gropp Lusk Skjellum 1999] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (MIT Press, Cambridge, MA, 1999), 2nd ed. 11

[Hamming 1973] R. W. Hamming, *Numerical Methods for Scientists and Engineers* (McGraw-Hill, New York, 1973), 2nd ed. 1.6

[Hodgkiss 1984] W. S. Hodgkiss, "An Oceanic Reverberation Model," IEEE J. Ocean. Eng. **OE-9**, 63-72 (1984). 10

[Hodgkiss 1989] W. S. Hodgkiss, "A Modular Approach to Exploratory Data Analysis," in *Proceedings Oceans '89, The Global Ocean* (IEEE Publication Number 89CH2780-5, 1989), pp. 1100-1104. 2.2

[Jackson Ivakin 1998] D. R. Jackson and A. N. Ivakin, "Scattering from elastic sea beds: First-order theory," J. Acoust. Soc. Am. **103**, 336-345 (1998). 11

[Katyl 2000] D. Katyl, "Distributed Systems at the Weapons Analysis Facility", Simulation Technology Magazine, Vol. 2, Issue 4b, June 28, 2000, http://www.sisostds.org/webletter/siso/Iss_62/. 1.2

[Knight 1981] W. C. Knight, R. G. Pridham, and S. M. Kay, "Digital Signal Processing for Sonar," Proc. IEEE **69**, 1451-1506 (1981). 1.1, 4.1.2

[Kulbago 1994] L. J. Kulbago, "A Study of Acoustic Backscatter from the Near-Surface Oceanic Bubble Layer," Master of Engineering thesis, The Pennsylvania State University, State College, PA, 1994. 5

[Lang Culver 1992] D. C. Lang and R. L. Culver, "A High Frequency Bistatic Ocean Surface Scattering Strength," ARL-PSU TM 92-342, Applied Research Laboratory, Pennsylvania State University, State College, PA, December 1992. 5

[Luby Lytle 1987] J. C. Luby and D. W. Lytle, "Autoregressive Modeling of Nonstationary Multibeam Sonar Reverberation," IEEE J. Ocean. Eng. **OE-12**, 116-129 (1987). 1.3, 10

[McDaniel 1990] S. T. McDaniel, "Models for Predicting Bistatic Surface Scattering Strength," ARL-PSU TM 90-88, Applied Research Laboratory, Pennsylvania State University, State College, PA, March 1990. 5

[Mackenzie 1959] K. V. Mackenzie, "Reflection of sound from coastal bottoms," J. Acoust. Soc. Am. **32**, 221-231 (1959). 6.4.3

[Mathematica] *Mathematica*, Wolfram Research, Inc., http://www.wolfram.com/products/mathematica. 2.2

[Matlab] *Matlab*, The MathWorks, Inc., http://www.mathworks.com/products/matlab/. 2.2

[Middleton 1967] D. Middleton, "A Statistical Theory of Reverberation and Similar First-Order Scattered Fields. Part I: Waveforms and the General Process," IEEE Trans. Inf. Theory **IT-13**, 372-392 (1967). 10

[Mitchell McPherson 1981] R. L. Mitchell and D. A. McPherson, "Generating Non-stationary Random Sequences," IEEE Trans. Aerospace Electron. Syst. **AES-17**, 553-560 (1981). 4.6.1, 4.6.1, 4

[Moe Jackson 1994] J. E. Moe and D. R. Jackson, "First-order perturbation solution for rough surface scattering cross section including the effects of gradients," J. Acoust. Soc. Am. **96**, 1748-1754 (1994). 5

[Mourad Dahl Jackson 1991] P. D. Mourad, P. H. Dahl, and D. R. Jackson, "Bottom Backscatter Modeling and Model/Data Comparison for 100-1000 Hz," APL-UW TR 9107, Applied Physics Laboratory, University of Washington, Seattle, WA, September 1991. 5

[Mourad Jackson 1989] P. D. Mourad and D. R. Jackson, "High Frequency Sonar Equation Models for Bottom Backscatter and Forward Loss," in *Proceedings OCEANS 89, The Global Ocean* (IEEE Publication Number 89CH2780-5, 1989), pp. 1168-1175. 5

[Mourad Jackson 1993] P. D. Mourad and D. R. Jackson, "A model/data comparison for low-frequency bottom backscatter," J. Acoust. Soc. Am. **94**, 344-358 (1993). 5

[Octave] *Octave*, University of Wisconsin, http://www.octave.org/. 2.2

[Ol-shevskii 1967] V. V. Ol-shevskii, *Characteristics of Sea Reverberation* (Consultants Bureau, New York, NY, 1967). 10

[Oppenheim Schafer 1989] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing* (Prentice-Hall, Englewood Cliffs, NJ, 1989). 1.1, 1.6, 4.5.1, 4.5.2, 4.5.2, 4.7

[Page-Jones 2000] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML* (Addison-Wesley, New York, NY, 2000). 1.1, 1.2

[Princehouse 1975] D. W. Princehouse, "REVGEN, A Real-time Reverberation Generator: Concept Development," APL-UW 7511, Applied Physics Laboratory, University of Washington, Seattle, WA, September 1975. 1.3, 10

[Princehouse 1978] D. W. Princehouse, "Reverberation Generator Algorithm, A Status Report," APL-UW 7806, Applied Physics Laboratory, University of Washington, Seattle, WA, February 1978. 1.3, 10

[Rouseff et al. 2001] D. Rouseff, D. R. Jackson, W. L. J. Fox, C. D. Jones, J. A. Ritcey, and D. R. Dowling, "Underwater Acoustic Communication by Passive-Phase Conjugation: Theory and Experimental Results," IEEE J. Ocean. Eng. **26**, 821-831 (2001) 1.3

[Sammelmann 2002] Gary Steven Sammelmann, "PC SWAT 7.0: Users' Manual," CSS Draft Report, Coastal Systems Station Code R22, Panama City, FL, February 2002. 1.2

[Schulten Anderson Gordon 1979] Z. Schulten, D. G. M. Anderson, and R. G. Gordon, "An Algorithm for the Evaluation of the Complex Airy Functions," J. Comp. Phys. **31**, 60-75 (1979). 5

[SST Web] R. P. Goddard, "The Sonar Simulation Toolset Web, Release 4.1", APL-UW TM 10-96 (Electronic Document), REVISED 2002, Applied Physics Laboratory, University of Washington, Seattle, WA, November 2002. 1.1, 1.3

[Stroustrup 2000] B. Stroustrup, *The C++ Programming Language* (Addison-Wesley, Reading, MA, 2000), Special Ed. 4.3

[Taylor 1955] T. T. Taylor, "Design of Line Sources Antennas for Narrow Beamwidth and Low Sidelobes," IRE Transaction on Antennas and Propagation **AP-3**, 16-28 (1955). Also in C. A. Balonis, *Antenna Theory — Analysis and Design* (Wiley & Sons, New York, 1997), pp. 358-362. 4

[TEAMS] *Undersea Warfare TEAMS*, led by ONR 333, ARL/PSU, and NUWC (Newport, RI), http://uswteams.arl.psu.edu/homepage/Homepage.svlt. 11

[Thorsos Broschat 1995] E. I. Thorsos and S. L. Broschat, "An investigation of the small slope approximation for scattering from rough surfaces. Part I. Theory," J. Acoust. Soc. Am. **97**, 2082-2093 (1995). 11

[Urick 1983] R. J. Urick, *Principles of Underwater Sound* (McGraw Hill, New York, NY, 1983), 3rd Ed. 1.1, 5, 6.3, 7.2

[Weinberg 1985] H. Weinberg, "Generic Sonar Model," NUSC TD 5971D, Naval Underwater Systems Center, New London, CT, June 1985. 5.3

[Weinberg Keenan 1996] H. Weinberg and R. E. Keenan, "Gaussian ray bundles for modeling high-frequency propagation loss under shallow-water conditions," J. Acoust. Soc. Am. **100**, 1421-1431 (1996). 1.3, 5.2

[Weinberg et al. 2001] H. Weinberg, R. L. Deavenport, E. H. McCarthy, and C. M. Anderson, "Comprehensive Acoustic System Simulation (CASS) Reference Guide", NUWC-NPT TM 01-016, Naval Undersea Warfare Center Division, Newport, RI, March 2001. 1.2, 5.2

[Williams Jackson 1998] K. L. Williams and D. R. Jackson, "Bistatic bottom scattering: Model, experiments, and model/data comparison," J. Acoust. Soc. Am. **103**, 169-181 (1998). 5

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2005	Technical Report

4. TITLE AND SUBTITLE
The Sonar Simulation Toolset, Release 4.1: Science, Mathematics, and Algorithms

5. FUNDING NUMBERS
ONR N00014-01-G-0460 and N00014-98-G-0001

6. AUTHOR(S)
Robert P. Goddard

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Applied Physics Laboratory
University of Washington
1013 NE 40th Street
Seattle, WA 98105-6698

8. PERFORMING ORGANIZATION REPORT NUMBER
TR 0404

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research, Code 333
Adam Nucci
800 N. Qunicy Street
Arlington, VA 22217-5660

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Sonar Simulation Toolset (SST) is a computer program that produces simulated sonar signals, enabling users to build an artificial ocean that sounds like a real ocean. Such signals are useful for designing new sonar systems, testing existing sonars, predicting performance, developing tactics, training operators and officers, planning experiments, and interpreting measurements. SST's simulated signals include reverberation, target echoes, discrete sound sources, and background noise with specified spectra. Externally generated or measured signals can be added to the output signal or used as transmissions. Eigenrays from the Generic Sonar Model (GSM) or the Comprehensive Acoustic System Simulation (CASS) can be used, making all of GSM's propagation models and CASS's Baussian Ray Bundle (GRAB) propagation model available to the SST user. A command language controls a large collection of component models describing the ocean, sonars, noise sources, targets, and signals. The software runs on several different UNIX computers. The software runs on several UNIX computers and Windows. SST's primary documentation is the SST Web (a large HTML "web site" distributed with the SST software), supported by a collection of documented examples.

This report emphasizes the science, mathematics, and algorithms underlying SST. This report is intended to be updated often and distributed with SST as an integral part of the SST documentation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
sonar, signal, reverberation, target echo, Generic Sonar Model (GSM), Comprehensive Acoustic System Simulation (CASS), Gaussian Ray Bundle (GRAB)	73
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR